THE UNIVERSITY OF SYDNEY

COPYRIGHT AND USE OF THIS THESIS

This thesis must be used in accordance with the provisions of the Copyright Act 1968.

Reproduction of material protected by copyright may be an infringement of copyright and copyright owners may be entitled to take legal action against persons who infringe their copyright.

Section 51 (2) of the Copyright Act permits an authorized officer of a university library or archives to provide a copy (by communication or otherwise) of an unpublished thesis kept in the library or archives, to a person who satisfies the authorized officer that he or she requires the reproduction for the purposes of research or study.

The Copyright Act grants the creator of a work a number of moral rights, specifically the right of attribution, the right against false attribution and the right of integrity.

You may infringe the author's moral rights if you:

- fail to acknowledge the author of this thesis if you quote sections from the work
- attribute this thesis to another author
- subject this thesis to derogatory treatment which may prejudice the author's reputation

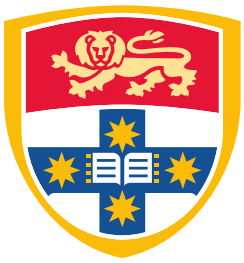
For further information contact the University's Director of Copyright Services

sydney.edu.au/copyright

Evaluating Parsers with Dependency Constraints

Dominick Ng

Supervisor: James R. Curran



THE UNIVERSITY OF SYDNEY

A thesis submitted

in fulfilment of the requirements

for the degree of Doctor of Philosophy at

The University of Sydney

School of Information Technologies

2016

Abstract

Many syntactic parsers now score over 90% on English in-domain evaluation, but the remaining errors have been challenging to address and difficult to quantify. Standard parsing metrics provide a consistent basis for comparison between parsers, but do not illuminate what errors remain to be addressed. This thesis develops a *constraint-based* evaluation for dependency and Combinatory Categorial Grammar (CCG) parsers to address this deficiency.

We examine the *constrained* and *cascading* impact, representing the direct and indirect effects of errors on parsing accuracy. This identifies errors that are the underlying source of problems in parses, compared to those which are a consequence of those problems. Kummerfeld et al. (2012) propose a static post-parsing analysis to categorise groups of errors into abstract classes, but this cannot account for cascading changes resulting from repairing errors, or limitations which may prevent the parser from applying a repair. In contrast, our technique is based on enforcing the presence of certain dependencies during parsing, whilst allowing the parser to choose the remainder of the analysis according to its grammar and model. We draw *constraints* for this process from gold-standard annotated corpora, grouping them into abstract error classes such as NP attachment, PP attachment, and clause attachment. By applying constraints from each error class in turn, we can examine how parsers respond when forced to correctly analyse each class.

We show how to apply dependency constraints in three parsers: the graph-based MSTParser (McDonald and Pereira, 2006) and the transition-based ZPar (Zhang and Clark, 2011b) dependency parsers, and the C&C CCG parser (Clark and Curran, 2007b). Each is widely-used and influential in the field, and each generates some form of predicate-argument dependencies. We compare the parsers, identifying common sources of error, and differences in the distribution of errors between constrained and cascaded impact. Our work allows us to contrast the implementations of each parser, and how they respond to constraint application.

Using our analysis, we experiment with new features for dependency parsing, which encode the frequency of proposed arcs in large-scale corpora derived from scanned books. These features are inspired by and extend on the work of Bansal and Klein (2011). We target these features at the most notable errors, and show how they address some, but not all of the difficult attachments across newswire and web text.

CCG parsing is particularly challenging, as different derivations do not always generate different dependencies. We develop *dependency hashing* to address semantically redundant parses in n-best CCG parsing, and demonstrate its necessity and effectiveness. Dependency hashing substantially improves the diversity of n-best CCG parses, and improves a CCG reranker when used for creating training and test data.

We show the intricacies of applying constraints to C&C, and describe instances where applying constraints causes the parser to produce a worse analysis. These results illustrate how algorithms which are relatively straightforward for constituency and dependency parsers are non-trivial to implement in CCG.

This work has explored dependencies as constraints in dependency and CCG parsing. We have shown how dependency hashing can efficiently eliminate semantically redundant CCG n-best parses, and presented a new evaluation framework based on enforcing the presence of dependencies in the output of the parser. By otherwise allowing the parser to proceed as it would have, we avoid the assumptions inherent in other work. We hope this work will provide insights into the remaining errors in parsing, and target efforts to address those errors, creating better syntactic analysis for downstream applications.

Acknowledgements

I am indebted to my supervisor, James Curran, who has been a mentor, inspiration, and guidance counsellor for eight years. It is thanks to James that I even contemplated a doctorate, and received so many opportunities to travel for my research. This work was driven by his never-ending stream of suggestions and questions, and refined through his review. Over the years that I have known him, he has never let me stop wondering about what the next opportunity might be.

My time as a PhD student was divided into three distinct phases. In the first, I worked closely with Mac Kim, Mark Johnson, Matthew Honnibal, and Daniel Tse, all of whom have left an imprint in this work. Thank you for your time, knowledge, and patience with me as an early researcher.

In the second phase, I was fortunate to be awarded a Fulbright Scholarship to spend a year at UC Berkeley, and this experience was life-changing. I would like to acknowledge the Berkeley NLP Group — Dan Klein, David Burkett, Greg Durrett, David Golland, David Hall, Taylor Berg-Kirkpatrick, and Jonathan Kummerfeld, who helped arrange the complexities of the visit — for a stimulating and thought-provoking year over the pond. I am particularly appreciative of Mohit Bansal, who provided great assistance once I returned home and strove to extend his work. My thanks also go to the Australian-American Fulbright Commission for granting me the scholarship and this time abroad.

Throughout my PhD, but particularly in the third phase, it has been a privilege to be a member of ə-lab, amongst outstanding friends and researchers. To James

Constable, Ben Hachey, Joel Nothman, Glen Pink, Will Radford, Nicky Ringland, and Kellie Webster, thank you for the many lunches, paper and thesis readings, the banter, the ideas, and the fun of being a group. This thesis was made infinitely better due to your comments, insights, advice, and presence. A particular acknowledgement must go to Tim Dawborn, who completed his thesis alongside me. Thanks for the many long days and nights spent co-commiserating in the lab over our respective products, code bugs, and formatting frustrations. Thanks also for keeping our servers alive under the weight of my experiments.

This thesis was wholly improved by the comments and feedback from many ə-lab members (thanks especially to Will, Kellie, Glen, and Joel), but also Dominic Balasuriya, who was kind enough to offer editorial assistance, and meticulous in combing out small grammatical mistakes.

A PhD is a long journey, and there are many unexpected twists and turns along the road. My sincerest gratitude to those friends who helped me when I stumbled along the way — most particularly Stephen Merity, Tara McIntosh, and James Kane. Thanks to my closest friends Sophie Liang, Bin Zhou, David Rizzuto, Jinna Kim, and Dom B for the good times and special moments. I'm also grateful for the philosophical discussions over many a game of squash with Jimmy Bai and Louis Gregory.

My parents, Alson and Esther, have always been there for me, as have my siblings Patrick and Samantha. We share a bond no matter the distance between us.

Finally, to Liz and Toby Toriola, thank you for everything. I love you both very much.

Statement of compliance

I certify that:

- I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;
- I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);
- this Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: *Dominick Ng*

Signature: Date: *15th March 2016*

Contents

1 Introduction **1**

1.1 Contributions 4

1.2 Outline 5

1.3 Publications Associated with this Thesis 6

2 Grammars, Corpora, and Evaluation **7**

2.1 Constituency Grammars 8

2.1.1 Constituency Trees 8

2.2 Dependency Grammars 11

2.2.1 Dependency Graphs 12

2.3 Combinatory Categorial Grammar 14

2.3.1 CCG Categories 15

2.3.2 Combinatory Rules 16

2.3.3 CCG Dependencies 21

2.3.4 Ambiguity in CCG 25

2.4 Corpora and Corpus Conversions 27

2.4.1 Constituency Grammar Corpora 27

2.4.2 Dependency Corpora 31

2.4.3 CCG Corpora 35

2.5 Parser Evaluation 40

2.5.1 Constituency Evaluation 40

2.5.2 Dependency Evaluation 41
2.5.3 CCG Evaluation 43
2.6 Summary 43

3 Constraint-based Error Analysis for Dependency Parsing 45
3.1 A Brief History of Computational Parsing 47
3.2 Dependency Parsing 51
3.2.1 MSTParser: Graph-based Dependency Parsing 51
3.2.2 ZPar: Transition-based Dependency Parsing 56
3.3 Analysing Parser Errors 58
3.4 Motivation 62
3.5 Constraint-driven Analysis of Parser Performance 64
3.5.1 Experimental Setup 65
3.5.2 Error Classes 66
3.6 Applying Constraints in Parsers 70
3.6.1 MSTParser implementation 70
3.6.2 ZPar implementation 72
3.7 Evaluation 74
3.7.1 Constrained and Cascaded Arcs 74
3.8 Comparing MSTParser and ZPar 77
3.8.1 Newswire 77
3.8.2 Web Text 85
3.9 MSTParser at Full Coverage 88
3.10 Summary 90

4 Surface and Syntactic *n*-gram Features for Dependency Parsing 93
4.1 Background 96
4.1.1 *N*-gram Corpora 96
4.1.2 Using ***n***-grams for Syntax 101

4.2 Features in MSTParser . . . 103
4.3 Surface *n*-gram Features . . . 104
4.3.1 First-order surface *n*-gram features . . . 105
4.3.2 Second-order surface *n*-gram features . . . 106
4.4 Syntactic *n*-gram Features . . . 107
4.4.1 First-order syntactic *n*-gram features . . . 108
4.4.2 Second-order syntactic *n*-gram features . . . 111
4.5 Experimental Setup . . . 111
4.5.1 Dependency schemes . . . 112
4.6 Results . . . 115
4.6.1 Surface *n*-gram Features . . . 116
4.6.2 Syntactic *n*-gram Features . . . 117
4.6.3 Combining Surface and Syntactic *n*-gram Features . . . 117
4.7 Analysis . . . 119
4.7.1 Corpora . . . 121
4.8 Constraint-based Evaluation . . . 122
4.9 Summary . . . 124

5 Dependency Hashing for CCG 127

5.1 Parsing with CCG . . . 128
5.2 The C&C Parser . . . 131
5.2.1 Supertagging . . . 131
5.2.2 Parsing CCG with the Cocke-Kasami-Younger Algorithm . . . 133
5.2.3 Decoding . . . 137
5.2.4 Parser Performance and Output . . . 139
5.2.5 Extensions to C&C . . . 141
5.3 Experimental Setup . . . 143
5.4 *n*-best Parsing and Reranking . . . 144
5.4.1 The *n*-best Algorithms of Huang and Chiang (2005) . . . 146

5.5 Dependency Hashing . . . 152
5.5.1 Hashing Implementation . . . 153
5.5.2 Hashing Performance . . . 159
5.5.3 Speed . . . 162
5.5.4 CCG reranking performance . . . 164
5.6 Summary . . . 165

6 Dependency Constraints for CCG **167**
6.1 CCG Constraints . . . 168
6.2 Implementation . . . 171
6.3 Creating Constraints . . . 176
6.3.1 Error Classes . . . 176
6.3.2 Constraints from CCGbank . . . 178
6.3.3 Constraint Statistics . . . 182
6.4 Summary . . . 183

7 Evaluating with CCG Constraints **185**
7.1 Evaluation Procedure . . . 186
7.2 Experimental Setup . . . 192
7.2.1 Configuration Examples . . . 193
7.3 Results . . . 196
7.3.1 Applying all constraints . . . 196
7.3.2 NP attachment constraints . . . 201
7.3.3 NP internal constraints . . . 202
7.3.4 Modifier attachment constraints . . . 203
7.3.5 Clause attachment constraints . . . 204
7.3.6 PP attachment constraints . . . 205
7.3.7 Coordination attachment constraints . . . 205
7.3.8 Root attachment constraints . . . 207

7.4 Constraints for Parser Debugging 208
7.4.1 Supertagger Errors 208
7.4.2 Non-standard CCG Rules in CCGbank 210
7.4.3 Inconsistencies between dependencies and the derivation 211
7.4.4 Co-indexation Inconsistencies in Categories 213
7.4.5 Categories Not Implemented in the Parser 215
7.5 Summary 215

8 Conclusion **219**
8.1 Future Work 221
8.2 Summary 222

Bibliography **223**

List of Figures

2.1 The list of dependencies depicted in the dependency tree in Parse 2.2. . 13

2.2 Fully annotated CCG categories. 22

2.3 The full set of CCG dependencies generated by the derivation in Parse 2.6. 24

2.4 The full set of CCG dependencies generated by the derivation in Parse 2.7. 25

2.5 The Penn Treebank bracketed string representation. 29

2.6 Different NP structures for a sentence from the Penn Treebank. 30

2.7 The grammatical relations hierarchy, reproduced from Briscoe (2006). . 33

2.8 The tabular dependency tree format. 34

2.9 The CCGbank string representation of a CCG derivation. 36

3.1 The construction of complete spans in Eisner's algorithm. 54

3.2 The construction of incomplete spans in Eisner's algorithm. 54

4.1 Raw 3-grams and their frequencies taken from Web1T. 97

4.2 Raw 3-grams and their frequencies taken from Google Books Ngrams. . 98

4.3 Raw 1-grams and their frequencies taken from Google Syntactic Ngrams. 99

4.4 Stanford attachment errors by gold argument POS tag. 120

5.1 A partially filled CCG chart. 134

6.1 A CCGbank tree with the path to the root bolded. 180

7.1 Baseline and constrained output for C&C. 188

7.2 Constrained and cascaded dependency sets for C&C. 191

List of Parses

2.1 A constituency tree for the sentence in Example 1. 9

2.2 A dependency tree for the sentence in Parse 2.1. 13

2.3 A dependency tree with prepositional coordination. 14

2.4 An example CCG tree using forward application. 17

2.5 The CCG type-raising combinator. 19

2.6 The CCG derivation of the sentence in Parse 2.1. 21

2.7 A CCG derivation with prepositional coordination. 24

2.8 Absorption ambiguity in CCG. 26

2.9 A CCG analysis of a relative clause. 38

3.1 MSTParser output compared to the gold-standard parse. 63

3.2 Constraining *root* to its correct position in MSTParser. 75

4.1 MSTParser's second-order factorisation. 104

4.2 Paraphrase-style context words in syntactic *n*-grams. 110

4.3 LTH and Stanford dependency analyses of a sentence. 113

6.1 A CCG derivation with a PP adjunct, adapted from Villavicencio (2002). 169

6.2 A CCG derivation with a PP argument. 169

6.3 A CCGbank tree with coordination. 179

7.1 A baseline CCG derivation with a PP argument. 187

7.2 A constrained CCG derivation with a PP adjunct. 187

7.3 A C&C parse corrected by applying constraints. 194
7.4 An incorrect C&C parse due to a missing supertagger category. 195
7.5 A C&C parse which is made worse by applying constraints. 209
7.6 The constrained analysis when applying *conj* constraints. 209
7.7 Non-equivalent coordination in CCGbank. 210
7.8 A CCG derivation that is inconsistent with a required dependency. . . . 212
7.9 A CCG derivation featuring subject control. 213
7.10 A CCG derivation where co-indexation blocks the correct parse. 214

List of Tables

3.1 Baseline MSTParser and ZPar parsing results over WSJ section 22. . . . 62

3.2 Constraints per error class over WSJ section 22. 68

3.3 Constraints per dependency label in Other attachments. 69

3.4 The impact of applying constraints to the sentence in Parse 3.1. 76

3.5 The results of applying constraints to MSTParser over WSJ 22. 78

3.6 The results of applying constraints to ZPar over WSJ 22. 78

3.7 Cascaded errors repaired per error class using NP constraints. 81

3.8 Cascaded errors repaired per error class using punctuation constraints. 84

3.9 The results of applying constraints to MSTParser over the Web Treebank. 85

3.10 The results of applying constraints to ZPar over the Web Treebank. . . . 86

3.11 Constraint evaluation for the baseline MSTParser. 89

4.1 Source text token counts for the *n*-gram corpora. 100

4.2 *n*-gram distribution by length in Web1T and Google Books. 100

4.3 First-order syntactic *n*-gram features and their counts. 109

4.4 Common and different unlabeled arcs between LTH and Stanford. . . . 114

4.5 LTH MSTParser surface and syntactic *n*-gram feature addition results. . 115

4.6 Stanford MSTParser surface and syntactic *n*-gram feature addition results. 116

4.7 LTH MSTParser test results with *n*-gram features. 117

4.8 Stanford MSTParser test results with *n*-gram features. 118

4.9 Comparing our combined features against the SANCL 2012 participants. 118

4.10 Surface n-gram queries missing in Web1T and Google Books Ngrams. . 121
4.11 Constraint evaluation for the best combined feature set. 123

5.1 Baseline C&C parser performance. 140
5.2 The default β, k, and other parameters used in the C&C parser. 144
5.3 Oracle n-best C&C parsing results. 152
5.4 Dependency diversity results for the n-best C&C parser. 153
5.5 GR diversity results for the n-best C&C parser. 153
5.6 Dependency hash collisions and comparisons over CCGbank section 00. 161
5.7 Dependency diversity results using dependency hashing. 162
5.8 GR diversity results using dependency hashing. 162
5.9 Oracle n-best C&C parsing results with dependency hashing. 163
5.10 Parsing speed with and without dependency hashing. 163
5.11 CCG reranking performance with and without dependency hashing. . . 165

6.1 CCG constraints per error class over CCGbank section 00. 183

7.1 C&C results for all constraints on automatic POS. 197
7.2 C&C results for NP attachment constraints on automatic POS. 201
7.3 C&C results for NP internal constraints on automatic POS. 202
7.4 C&C results for modifier constraints on automatic POS. 203
7.5 C&C results for clause constraints on automatic POS. 204
7.6 C&C results for PP attachment constraints on automatic POS. 205
7.7 C&C results for coordination constraints on automatic POS. 206
7.8 C&C results for root constraints on automatic POS. 207

List of Algorithms

3.1 *Eisner's Algorithm* . . . 55

3.2 *Transition-based Beam Search Parsing* . . . 58

3.3 *constrained-label* . . . 71

3.4 *Constrained Eisner's Algorithm* . . . 71

3.5 *ConstrainedArcLeft* . . . 72

3.6 *ConstrainedArcRight* . . . 73

5.1 **CKY CCG** *Parsing* . . . 137

5.2 *decode*$_{\text{nf}}$ . . . 138

5.3 *best-equiv* . . . 138

5.4 *best-score* . . . 139

5.5 *mult*$_{\text{n}}$ . . . 147

5.6 *merge*$_{\text{n}}$ . . . 147

5.7 *new-mult*$_{\text{n}}$ . . . 148

5.8 *find-best*$_{\text{n}}$ . . . 149

5.9 *get-candidates*$_{\text{n}}$ . . . 150

5.10 *lazy-best*$_{\text{n}}$ . . . 150

5.11 *lazy-best-equiv*$_{\text{n}}$ . . . 151

5.12 *lazy-next*$_{\text{n}}$ . . . 151

5.13 *hash-rule* . . . 156

5.14 *hash-category* . . . 156

5.15 *hash-mult*$_{\text{n}}$ . . . 157

5.16 ***hash-merge***$_{\mathrm{n}}$ . . . 158
5.17 ***new-hash-mult***$_{\mathrm{n}}$ . . . 159
5.18 ***hash-find-best***$_{\mathrm{n}}$ . . . 159
5.19 ***hash-get-candidates***$_{\mathrm{n}}$ . . . 160
5.20 ***lazy-best-equiv***$_{\mathrm{n}}$ . . . 160
5.21 ***lazy-next***$_{\mathrm{n}}$ . . . 161
6.1 ***load-constraints*** . . . 171
6.2 ***decode-constrained***$_{\mathrm{nf}}$ . . . 172
6.3 ***best-equiv-constrained*** . . . 173
6.4 ***best-score-constrained*** . . . 174
6.5 ***check-satisfaction*** . . . 175
6.6 ***violates-variables*** . . . 175
6.7 ***extract-conj-deps*** . . . 180
6.8 ***extract-root-deps*** . . . 181
6.9 ***apply-root-constraints*** . . . 181
7.1 ***make-dep-sets*** . . . 190

1 Introduction

Identifying syntactic structure in language is important for many downstream applications in computational linguistics. Accurate parsing has been the target of research for decades, and state-of-the-art English parsers now score over 90% on standard newswire metrics across different formalisms (Charniak and Johnson, 2005; McClosky et al., 2006; Petrov and Klein, 2007; Zhang and Clark, 2011b; Zhang and McDonald, 2014). However, the remaining errors have been challenging to address and difficult to quantify. Standard parsing evaluation metrics provide consistent measures of performance between different parsers, but do little to identify the kinds of errors each parser makes. They also cannot identify whether groups of errors have a common underlying cause, even though such diagnostic evaluation is crucial for improving parsing performance.

Kummerfeld et al. (2012) proposed a evaluation procedure to analyse the errors made by constituency parsers, and group them into abstract higher-level classes, such as incorrect noun phrase, prepositional phrase, or clause attachments. They group each error class based on the operations that repair each error — movement, addition, and deletion of nodes in the tree structure. They use this to statically analyse the output of several parsers and classify their errors. The weakness of this approach is that when parsers are run with constraints dynamically, there is no guarantee that other sections of the parse will remain the same. Higher-order features, structural constraints, and the parser model can all trigger cascading changes based on a single different attachment. These cascading changes may repair other errors in the tree, or they may create new errors where there were previously none.

In this thesis, we propose a constraint-based evaluation procedure using dependencies. We group gold-standard dependency labels into error classes, and force parsers to produce the attachments in each error class. For the remainder of the parse, we allow each parser to produce the most probable analysis under its grammar and model, without making assumptions about its behaviour. We investigate the *constrained* impact produced directly by applying dependencies as constraints, as well as the *cascaded* impact induced elsewhere in the tree. Our approach avoids the issues of Kummerfeld et al. (2012) by not interfering with the parser's decoding other than enforcing the presence of constrained arcs. In this way, we establish a sound basis for diagnosing parsing errors as being sources or consequences of higher-level issues in the tree.

We implement our procedure for MSTParser (McDonald and Pereira, 2006) and ZPar (Zhang and Clark, 2011b), two widely-used dependency parsers based on two different and popular representations of the task. We also implement our procedure for the C&C parser (Clark and Curran, 2007b), an efficient Combinatory Categorial Grammar (CCG; Steedman, 2000) parser which produces predicate-argument dependencies as a primary output. We describe the modifications required for each parser to apply constraints, and compare the characteristics of each parser's performance using constraints.

Our evaluation found that NP and PP attachments are particular challenges across all parsers, but affect parsing accuracy in different ways. For dependency parsers, PPs are relatively localised errors with limited influence, but incorrect NP attachment causes *cascading error* through the remainder of the parse, causing further misattachments. The opposite is true for evaluation in CCG parsing due to the presence of head categories in dependencies: NP attachments are more localised, while PP attachments cause substantial cascading errors.

Punctuation errors are frequently ignored in parser evaluation (Yamada and Matsumoto, 2003; Buchholz and Marsi, 2006), and have limited or inconsistent treatment in corpora. However, incorrect punctuation is a strong indicator of an erroneous dependency parse, causing substantial cascading impact in both MSTParser and ZPar.

CCG parsing is particularly challenging, as different derivation structures do not necessarily generate unique semantic structures. Algorithms developed on constituency and dependency parsers are more difficult to apply to CCG. We illustrate how our constraint evaluation is non-trivial to implement for CCG, necessitating special care with the logical form that underpins all CCG analyses. We show how applying constraints serves as an evaluation not only of CCG parser performance, but also the implementation of the parser itself. Experiments with *n*-best CCG parsing show that it is another task where different derivations generating the same dependencies affects performance; we find that *n*-best parsing algorithms designed without CCG's dependency formulation in mind are insufficient, producing considerable numbers of semantically redundant parses. We design *dependency hashing* as an efficient solution to this issue, and demonstrate how it improves a CCG reranker.

Features from large unannotated corpora have been shown to assist with NP and PP attachments (Volk, 2001; Nakov and Hearst, 2005a), which our constraint-based evaluation identified as being large sources of errors for parsers. Building on the surface *n*-gram features of Bansal and Klein (2011), we attack these errors for MSTParser by experimenting with higher-order features based on different *n*-gram sources. We develop new features based on syntactic *n*-grams, subtree fragments from parsed English books, and compare their effectiveness with surface *n*-gram features.

All our feature types perform similarly in isolation across dependency schemes and domains, and are also complementary; a combined system of surface and syntactic *n*-gram features outperforms all other feature types in all configurations and across domains, achieving up to 1.3% UAS improvements in-domain and 1.6% out-of-domain. The features are also successful at addressing some of the largest error classes, specifically NP and PP attachments. However they do not fundamentally change the overall error distribution, and primarily serve to correct constrained errors rather than reduce cascaded impact.

1.1 Contributions

We describe how dependency-level constraints can be applied to graph-based and transition-based dependency parsers for evaluation. By grouping dependency labels into meaningful error classes, we can applying each group in turn and examine the accuracy of the parser with respect to the constraints, and their cascading impact through a sentence. This technique allows us to identify the relative constrained and cascaded impact of each error class, and provides an informative platform for comparing dependency parser performance.

Our in-depth investigation of dependency parser performance shows how the transition-based ZPar outperforms the graph-based MSTParser, despite examining only a fraction of the entire search space. We also show how ZPar is better able to use constraints to create cascading improvements elsewhere in the parse.

We develop dependency parsing features for MSTParser using the Google Books Ngrams and Google Syntactic Ngrams corpora, which are large-scale *n*-gram resources extracted from scanned English books. We show how syntactic features, despite being better aligned to the task, perform no better than surface features on in-domain text. However, syntactic features improve accuracy out of domain. The two features types also stack; we achieve up to 1.3% accuracy improvements on newswire, and 1.6% on web text, significantly more than either type in isolation.

Our examination of *n*-best CCG parsing reveals that the standard *n*-best parsing algorithms are deficient, as they do not account for different CCG derivations potentially generating the same dependencies. We describe *dependency hashing* to eliminate this issue, and implement new *n*-best CCG parsing algorithms using this technique. We demonstrate the effectiveness of dependency hashing in efficiently generating distinct CCG parses without costly semantic equivalence checks, and show how this leads to improved CCG reranking performance.

We take our constraint evaluation procedure for dependency parsers and implement it for the C&C CCG parser. We describe the complexities of the implementation, and the attributes of CCG which make applying such constraints difficult. Applying a similar evaluation for CCG reveals broad similarities in error distribution to dependency parsers, but also allows us to investigate the implementation of C&C itself.

In summary, this work demonstrates the utility of diagnostic evaluation without assumptions about parser behaviour. We illustrate how different error classes behave differently in parsers, beyond the simple metrics of attachment accuracy. Some errors cause much more cascading impact than others, suggesting that they should be prioritised in further efforts to address parser performance.

1.2 Outline

We begin by surveying the grammars, corpora, and evaluation procedures used in this thesis in Chapter 2, contrasting the features of constituency, dependency, and Combinatory Categorial Grammars across these areas.

Chapter 3 continues with an examination of parsing, focusing on graph-based and transition-based techniques. We describe the implementation of our constraint-based evaluation procedure, and compare the results of applying constraints to MSTParser and ZPar.

In Chapter 4, we take MSTParser, and experiment with surface and syntactic n-gram features extracted from Google Books. We evaluate the features with our constraint-based procedure from Chapter 3.

We turn our focus from dependency parsing to CCG parsing in Chapter 5, examining the inadequacy of standard n-best parsing algorithms for CCG. We propose dependency hashing as a solution, and demonstrate its correctness and effectiveness. We use the distinct parses to improve a CCG reranker for English.

Chapters 6 and 7 unify our work on dependency parsing and CCG parsing by implementing constraints on CCG dependencies in the C&C parser. We discuss the relative complexities of this process compared to the dependency parsers, and show how constraints can be used not only for analysing a CCG parser's performance, but also the implementation of the parser itself.

Finally, Chapter 8 discusses the avenues for future work, and summarises the core findings of this thesis.

1.3 Publications Associated with this Thesis

The constraint-based error analysis technique for dependency parsers described in Chapter 3 was published as *Identifying Cascading Errors using Constraints in Dependency Parsing* at the 53rd Annual Meeting of the Association for Computational Linguistics (Ng and Curran, 2015).

The work on surface and syntactic *n*-gram features for dependency parsing in Chapter 4 was made available on arXiv as *Web-scale Surface and Syntactic* n*-gram Features for Dependency Parsing* (Ng et al., 2015).

The dependency hashing technique described in Chapter 5 was published as *Dependency Hashing for n-best* CCG *Parsing* at the 50th Annual Meeting of the Association for Computational Linguistics (Ng and Curran, 2012).

Our *n*-best CCG parser using dependency hashing was used for the CCG reranker published as *Improving Combinatory Categorial Grammar Parse Reranking with Dependency Grammar Features* at the 24th International Conference on Computational Linguistics (Kim et al., 2012).

2 Grammars, Corpora, and Evaluation

All natural languages are structured, and it is this structure that allows people to accurately communicate. The structure of language is referred to as *syntax*, and it describes how the *surface form* words interact with one another to form sentences with meaning. For example,

(1) *The bank employs 8,000 people in Spain and 2,000 abroad*; and

(2) *The bank employs 2,000 people in Spain and 8,000 abroad*
convey different meanings using the same words, and

(3) *8,000 people in Spain and 2,000 abroad are employed by the bank*
implies the meaning of Example 1.

All three example sentences convey that the bank employs 10,000 people in total. Examples 1 and 3 place 8,000 of those people in Spain, with 2,000 in other countries. Example 2 reverses the distribution, with 2,000 in Spain and 8,000 elsewhere.

Natural language processing (NLP) is the branch of artificial intelligence concerned with the automatic interpretation of human utterances and records. Within NLP, *syntactic parsing* recognises the structure of language and identifies how words interact and fit together according to some underlying *grammar*.

Language typically exhibits *compositional semantics*, whereby the meaning of a sentence is derived from the meanings of its components. Language is inherently *syntactically ambiguous*, as words many be combined in many different ways. Accurate

parsing is crucial for resolving these ambiguities for downstream applications. Parsing is one of the oldest and most deeply researched areas of NLP. From its earliest days as a rule-driven process to today's strongly statistical focus, parsing has grown with NLP and remains a common component of many language processing pipelines that seek a deeper understanding of textual meaning.

Research in parsing is intimately entwined with the underlying grammars and annotated data used for training and testing. This work spans across a number of grammar formalisms for English, and so we begin by discussing grammars, the corpora based on them, and the evaluation procedures for parsers using the corpora.

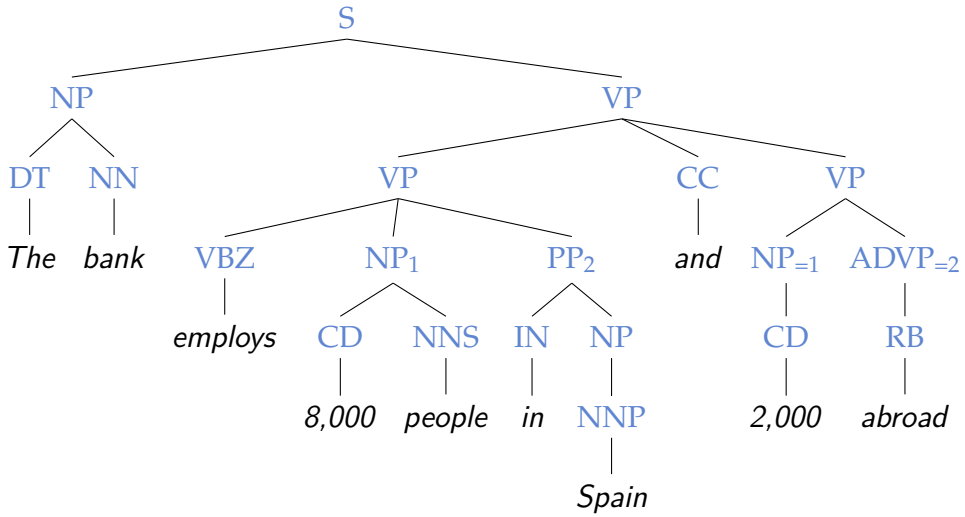
2.1 Constituency Grammars

A *grammar* defines a set of rules describing the syntax of a language. These rules may be used to determine whether text belongs to the language or not. They are also used by *parsers* to construct *derivations* of sentences under the grammar, commonly depicted in the form of a *syntax tree* or *dependency tree* illustrating how subcomponents of the analysis interact with each other.

Grammars vary in their expressiveness and completeness. In this thesis, we utilise data originally created using an English *constituency grammar*, but we work primarily with *dependency grammars* and *Combinatory Categorial Grammar*. We now present a broad background of these three classes of grammar, and describe their properties.

2.1.1 Constituency Trees

Constituency grammars decompose sentences into hierarchical groupings of smaller constituents. Each constituent, also known as a *phrase*, represents a cohesive unit of syntax. These phrases are progressively combined together into larger units in a tree structure that spans the entire sentence. Parse 2.1 depicts an example constituency tree



Parse 2.1: A constituency tree for the sentence in Example 1.

for the sentence in Example 1 using the OntoNotes annotation (discussed further in Section 2.4.1).

Constituency grammars work well for languages with fairly rigid word order, such as English. The smallest typical constituents in English are individual words. These are labeled with a *part-of-speech* (POS) tag specifying their grammatical form or function (such as *noun*, *verb*, or *preposition*). These tags are then used in *context-free production rules* that describe how words of each form combine into progressively larger *constructions*. The defining difference between any two constituency grammars is the set of these rules that is used; different grammars may combine constituents in different orders or specify completely different constructions. Some examples of the context-free production rules used in Parse 2.1 include:

$$\begin{aligned} \text{S} &\rightarrow \text{NP VP} \\ \text{NP} &\rightarrow \left\{ \begin{matrix} \text{DT JJ}^* \text{ NN} \\ \text{NN} \end{matrix} \right\} \\ \text{VP} &\rightarrow \text{VBZ NP PP} \\ \text{PP} &\rightarrow \text{IN NP} \end{aligned}$$

where S = sentence, NP = noun phrase, DT = determiner, JJ = adjective, NN = noun, VP = verb phrase, VBZ = singular third-person verb, PP = prepositional phrase, and IN = preposition.

The application of production rules produces a syntax tree, where the left hand side of each rule is a subtree root, and the right hand side of the rule are that root's children. These rules effectively link each word and phrase in a sentence with zero or more closest siblings and its parent.

Constituency grammars group the words of a sentence into the clusters with which they interact most closely, and are sometimes termed *context-free grammars* (CFGs). However, the context-free property requires that constructions only contain consecutive words in the sentence. This is insufficient to explain many phenomena in natural language. For example, in Parse 2.1, the *object* of the main verb phrase is coordinated: the sentence implies both *employs 8,000 people in Spain* and *employs 2,000 people abroad*. The verb phrase headed by *employs* has a parallel structure over the coordination in a process known as *gapping*. This is a problem in the constituency tree, as Parse 2.1 shows that the components *8,000 people in Spain* and *2,000 abroad* are not equivalent, despite both functioning as an object of the verb. The latter is termed a *non-constituent*, as it has a 'hole' where a verb is required to form a phrase. Such constituents cannot be coordinated together.

There are two possible solutions to this issue. The first is to include rules in the grammar that allow such non-equivalent (but functionally identical) components to be coordinated. This is unappealing, as it would lead to a large proliferation of rules to account for all of the possible ways in which this kind of coordination could occur. The second solution is to employ markers in the tree to indicate the movement of constituents. These markers take the form of *co-indexation variables*, *traces*, and *null elements*, which indicate where missing elements occur, and where in the sentence they were actually placed. This approach is typically used in constituency corpora, as it

avoids rule proliferation and enables the recovery of the underlying predicate-argument structure (discussed in further detail below).

In Parse 2.1, the verb *employs* is placed in a phrase with *8,000 people in Spain*, where the NP *8,000 people* is assigned the variable 1. This is then co-indexed with the same variable on *2,000*, indicating that the second component reuses the structure from the first. Similarly, the modifier *in Spain* is co-indexed with *abroad* using variable 2. This allows the gapped second fragment to be associated with its verb.

2.2 Dependency Grammars

The need for co-indexed variables to indicate the gapped shared structure in Parse 2.1 illustrates one of the primary issues with constituency grammars: that the phrase structure of a sentence alone is not always a sufficient proxy for the functional relationships between the words. *Dependency grammars* abstract away from the phrase structure, and focus on these functional relationships. This is particularly useful for languages with more flexible word order, where constituency grammars are cumbersome.

In a constituency grammar, each phrase has a *head* that indicates its semantic or functional core. Phrases are typically named for the grammatical form of their head: e.g. noun phrases are headed by nouns, and verb phrases are headed by verbs. The non-head components of a phrase are *dependents* of the head; that is, their presence in the sentence is as a *modifier* or a *complement* to the head. Thus, each phrase contains a number of head-dependent relationships, which can be termed as *dependencies*.

Dependencies are directed binary relations between a head and dependent word in a sentence, with a *label* describing the relationship between the head and dependent, e.g. a verb to noun dependency may be labelled as a *subject* or *object* relation. Each head may have multiple dependents, and one word (typically the main verb) is denoted as the *root*, or the head of the entire sentence.

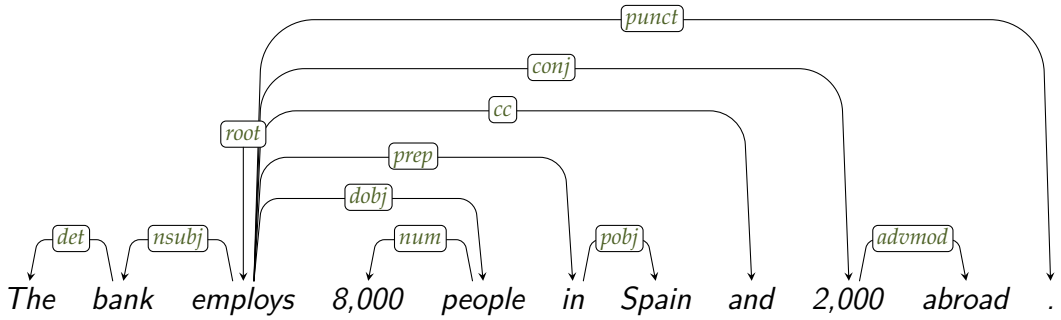
Whilst we have described dependencies in terms of heads of phrases, it is important to note that they are not constrained to the phrase structure of a sentence, and may be created between any pair of words in a sentence. Dependency grammars differ in the labels they assign to dependencies, and how they define which words are heads and which are dependents. The various choices and linguistic motivations for those choices have given rise to several different dependency grammars for English. In this thesis, we work most closely with the grammar described in Johansson and Nugues (2007) (known as LTH), and the basic variant of the Stanford dependency grammar described in de Marneffe and Manning (2008b). We will discuss both of these schemes in more detail in Section 2.4.2.

2.2.1 Dependency Graphs

A *dependency graph* illustrates the dependency analysis of a sentence. Typically, the words are depicted in sentence order, and the dependencies are indicated by *arcs* issuing from the head word and pointing to the argument word. Depending on the properties of the dependency grammar used for an analysis, this graph may take the form of a *dependency tree*, a directed acyclic connected graph where each word has a single head. The grammar may specify a *root* node, an invisible addition to the sentence which serves as the head of any (or the only) word without a head. The graph may reveal the grammar to be *projective*, where no dependency arcs cross over one another, or *non-projective*, where there are crossing arcs in the graph.

Parse 2.2 depicts a dependency tree for the sentence in Parse 2.1 using the basic variant of Stanford dependencies. The depicted sentence is projective, with the grammar requiring each word in the sentence to have a single head, with a single root word overall. Some of the labels used in this grammar include *det* (determiner), *nsubj* (nominal subject), *dobj* (direct object), and *pobj* (prepositional object).

Instead of a graphical representation, a dependency analysis may also be represented as a list of the head-dependent relationships (Figure 2.1).

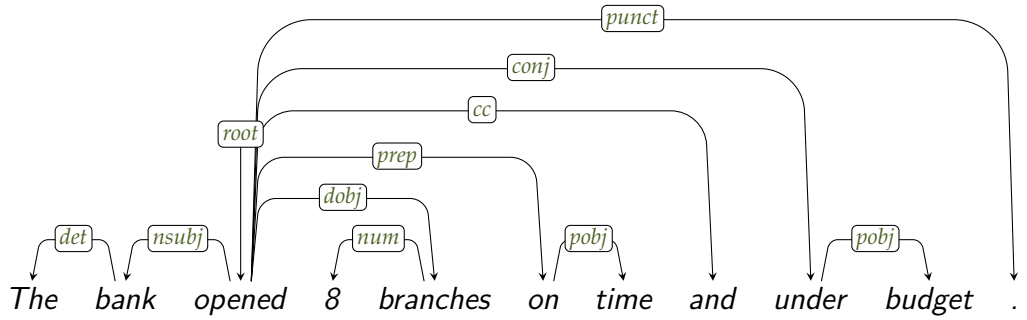


Parse 2.2: A dependency tree for the sentence in Parse 2.1.

⟨*det bank The*⟩	⟨*cc employs and*⟩
⟨*det bank The*⟩	⟨*conj employs 2,000*⟩
⟨*nsubj employs bank*⟩	⟨*num people 8,000*⟩
⟨*dobj employs people*⟩	⟨*pobj in Spain*⟩
⟨*prep employs in*⟩	⟨*advmod 2,000 abroad*⟩

Figure 2.1: The list of dependencies depicted in the dependency tree in Parse 2.2.

Unlike the constituency grammar, which associated words most closely with their siblings in each phrase, the dependency grammar directly associates each word with its closest functional partner — which may or may not be close to it in the sentence. For example, the coordinator *and* is headed by *employs* in the dependency tree, whilst it lies in a separate phrase in the constituency tree. The main verb, *employs* is the root of the sentence, with the noun *bank* being the subject. The phrases *8,000 people* and *2,000 abroad* are directly linked to *employs*, showing a different approximation of the coordinated VP structure. However, there is no longer any indication of the parallel structure of the coordination: the PP *in Spain* attaches directly to *employs*, but the modifier *abroad* does not.



Parse 2.3: A dependency tree that is isomorphic to Parse 2.2, but features prepositional coordination rather than verb coordination.

2.3 Combinatory Categorial Grammar

Whilst the dependency grammar analysis has clearly shown the coordination of the noun phrases *8,000 people in Spain* and *2,000 abroad*, the restriction of one head per word in this particular scheme introduces potential ambiguities. Parse 2.3 depicts a dependency representation of a similar sentence to Parse 2.2, except the object coordination has been replaced with a prepositional phrase coordination (of *on time* and *under budget*). The two trees are isomorphic to each other, with only one label different between the two. Whilst it is possible to infer the coordinated phrases from the label change and the POS tags assigned to the words, it is not immediately clear from the tree which case is which. Other phenomena such as right node raising and control are also challenging to represent with one head per word (Johansson and Nugues, 2007).

Combinatory Categorial Grammar (CCG; Steedman, 2000) is a linguistically-motivated grammar formalism based on formal logic. CCG is a lexicalised grammar; each word is assigned a *lexical category* that describes its syntactic behaviour in the sentence. Categories differ from POS tags in that they contain far more information about the word's behaviour; the production rules used to specify the behaviour of a constituency grammar are encoded directly within the category. The grammar can then use simple rules to allow categories to combine together to form an analysis of a sentence.

Categories also encode a semantic interpretation that is applied as they are combined together in an analysis. This gives rise to a *transparent* interface between the syntax of a sentence (represented by the successive category combinations) and its semantic interpretation (Steedman, 2000). This transparency has driven recent work in semantic analysis using CCG, particularly in semantic parsing (Kwiatkowski et al., 2011; Artzi et al., 2014) question-answering (Kwiatkowski et al., 2013; Reddy et al., 2014) and distributional semantics (Lewis and Steedman, 2013).

2.3.1 CCG Categories

CCG categories may be *atomic* or *complex*. Atomic categories represent constituents that are syntactically complete, such as S (sentence), N (noun), NP (noun phrase), and PP (prepositional phrase).

Complex categories represent constituents as *functors* that require *arguments* to become syntactically complete. They are recursively constructed from other categories and take the form $X\backslash Y$ or X/Y, where X and Y may in turn be atomic or complex. X is the result category produced when the complex category is combined with the argument Y. The slash directionality indicates whether Y is expected to the right (forward slash) or left (backward slash) of the category respectively. For example, English intransitive verbs consume a noun phrase to the left (typically denoted the *subject*) to form a sentence:

$$\textit{sneezes} := S\backslash NP$$

Multiple arguments are specified using nesting. English transitive verbs consume one noun phrase to the right (the *object*) and one noun phrase to the left (the *subject*) to form a sentence. In other words, they take one noun phrase to the right to form an intransitive verb:

$$\textit{employs} := (S\backslash NP)/NP$$

Ditransitive verbs consume one additional noun phrase to the right (a second object) and produce a transitive verb:

$$\textit{told} := ((S\backslash NP)/NP)/NP$$

Modifier categories consume a category as an argument, and return the same category as a result. For example, adverbs modify intransitive verbs, and adjectives modify nouns as follows:

$$\textit{quickly} := (S\backslash NP)\backslash(S\backslash NP)$$

$$\textit{large} := N/N$$

All CCG categories include head information in the form of variables. At the word level, each word is its own head. For words that have complex categories, the head of the result is indicated using variables on the subparts of each argument category. For example, the word *the* bears the category $(NP_Y/N_Y)_{\textit{the}}$. This indicates that the word is its own head, and that it combines with a word to the right with category N and head Y to product a constituent with category NP and head Y. In Parse 2.4, the head of the noun phrase *the bank* is *bank*.

Words performing the same syntactic function would ideally take the same category at all times. However, ambiguity means that some words may be assigned different categories in different situations. We will return to the problems posed by ambiguity in CCG later in this work. For example, *homonyms* with identical lexical form, but different syntactic functions must take different categories in their different roles:

$$\textit{row (c.f. pew)} := N$$

$$\textit{row (c.f. boat)} := (S\backslash NP)/NP$$

2.3.2 Combinatory Rules

Categories are combined together using a small set of *combinatory rules* to form a *derivation*. In contrast to constituency grammars, where subcategorisation information

Parse 2.4: An example CCG tree using the forward application combinatory rule with the determiner *The*. The tree is equivalent to the derivation depicted to the right, which is the conventional presentation of CCG analyses.

is expressed in the context-free rules of the grammar, CCG categories directly specify the expected syntactic structure of their arguments, and thus the grammar requires few rules. Combinatory rules describe how complex categories may be combined with their arguments (application), composed together in a currying fashion (composition), or changed into functors (type-raising). These rules reflect the influence of formal logic on CCG as they can be described in terms of functors acting upon arguments.

The most common combinatory rules in CCG are *function application rules*, which describe the simple process of a complex category consuming its outermost argument to form its result category. *Forward* ($>$) and *backward* ($<$) *application* respectively refer to consuming an argument to the right and left of a complex category, denoted with:

$$X/Y \;\; Y \;\; \Rightarrow \;\; X \quad (>)$$

$$Y \;\; X\backslash Y \;\; \Rightarrow \;\; X \quad (<)$$

Parse 2.4 depicts a CCG tree on the left, and its equivalent *derivation* convention on the right. If viewed upside down, the derivation is a tree, where each category corresponds to a node. Each step of the derivation depicts the rule application combining two categories together, eventually building the final analysis. The binary branching trees mean that CCG analyses are typically much deeper than constituency trees.

Forward and backward application may be viewed as abstractions over the entire space of context-free production rules in constituent grammars. Whilst each production rule must be explicitly specified for it to be used in a constituency grammar, the

specification is pushed into the category level in CCG. This allows the rules to operate independently of the lexicon and the grammar, dramatically reducing redundancy.

The category structure combined with forward and backward application rules alone form the *Categorial grammar*, or *AB grammar*, after Adjukiewicz (1935) and Bar-Hillel (1953). Bar-Hillel et al. (1960) proved that this grammar is weakly equivalent to a CFG, as both are capable of generating the same formal languages.

Parse 2.6 depicts a CCG derivation of a sentence, introducing several additional combinatory rules. These rules increase the expressive power of the grammar from context-free to mildly context-sensitive (Vijay-Shanker and Weir, 1994), and they allow the grammar to account for a diverse range of linguistic phenomena through rebracketing, such as the non-constituent coordination Example 1 which causes problems for constituency and dependency grammars.

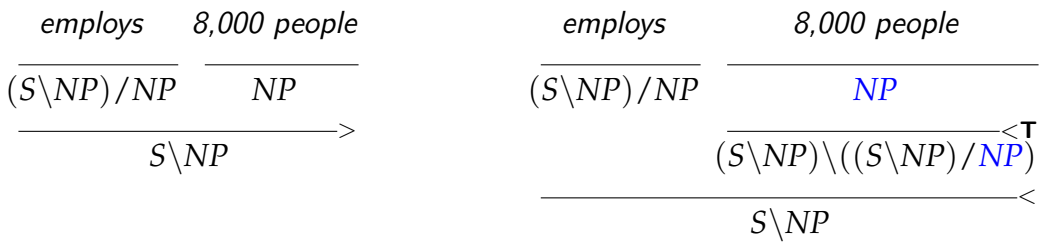
The first new rule introduced is *coordination*, which allows two equivalent constituents separated by a *conj* category (e.g. *and*) to be combined together to form a new constituent with the same category:

$$X \ \mathit{conj} \ X \Rightarrow X \quad (<\mathbf{\Phi}>)$$

The challenge remains to transform the non-equivalent constructions *8,000 people in Spain* and *2,000 abroad* into entities which can be coordinated. The key observation is that each construction is effectively looking for a transitive verb to its left, and upon finding one will produce an intransitive verb. However, application rules alone cannot produce this category from those assigned to the words:

$$\dfrac{\dfrac{\mathit{employs} \quad \mathit{8{,}000\ people}}{(S\backslash NP)/NP} \quad \dfrac{\mathit{in} \quad \mathit{Spain}}{((S\backslash NP)\backslash(S\backslash NP))/NP} \quad \dfrac{}{NP}}{\dfrac{}{(S\backslash NP)\backslash(S\backslash NP)}>}}{\dfrac{}{(S\backslash NP)\backslash((S\backslash NP)/NP)}>?}$$

Consider the noun phrase *8,000 people*. Its category is the atomic NP, which does not specify any arguments, and so can only be consumed by a complex category.



Parse 2.5: The CCG type-raising combinator transforms a category (here an *NP*) into a category consuming the category that would have originally consumed it.

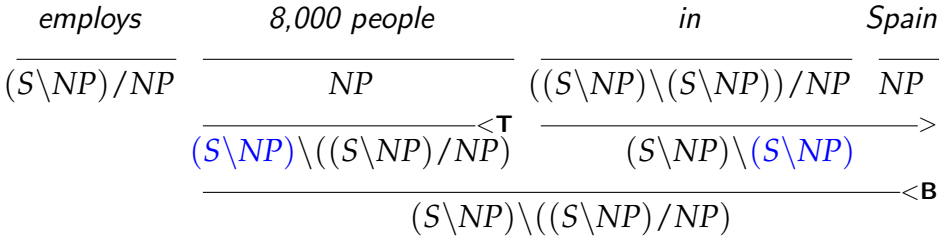
However, the CCG *type-raising* rule allows the category to be rewritten as a complex category by effectively imprinting the behaviour of a complex category onto it. It becomes a category that consumes the category that would have consumed it originally. For example, when an *NP* is in object position, it can be thought of as a constituent that can consume a transitive verb to its left to produce an intransitive verb

The fragment on the left in Parse 2.5 depicts the typical behaviour of a transitive verb consuming the object *NP*. The fragment on the right *type-raises* the *NP*, allowing it to consume the transitive verb instead to produce the same result in a non-canonical way. The blue *NP* argument in the type-raise result is the original category. More generally, the type-raising rules are defined as follows, with forward and backward variants specifying on which side the type-raised category is expecting its argument:

$$X \Rightarrow T/(T\backslash X) \quad (>\mathbf{T})$$

$$X \Rightarrow T\backslash(T/X) \quad (<\mathbf{T})$$

Alone, type-raising simply generates derivational ambiguity. However, type-raising combined with the *composition* rules allow for constituents to be rebracketed:



Composition has allowed the outermost categories (coloured blue) to be the factors upon which combination can occur. The four composition rules allow categories to combine when the result of one category is the same as the argument of the other. *Forward* composition calls for the argument of the left category to be the same as the result of the right category, while *backward* composition requires the result of the left category to be the same as the argument of the right category. *Harmonic* composition is *order preserving* as the input categories must still occur in canonical order for the combination to succeed. *Crossed* composition is *order permuting*, and is used to analyse constructions such as heavy NP shift in English, where an object NP is sufficiently large enough to swap positions with an intervening PP (e.g. *gave to the bank an extremely large and valuable ancient artefact*).

Forward composition

$$X/Y \;\; Y/Z \;\; \Rightarrow \;\; X/Z \quad (>\mathbf{B})$$

Backward composition

$$Y\backslash Z \;\; X\backslash Y \;\; \Rightarrow \;\; X\backslash Z \quad (<\mathbf{B})$$

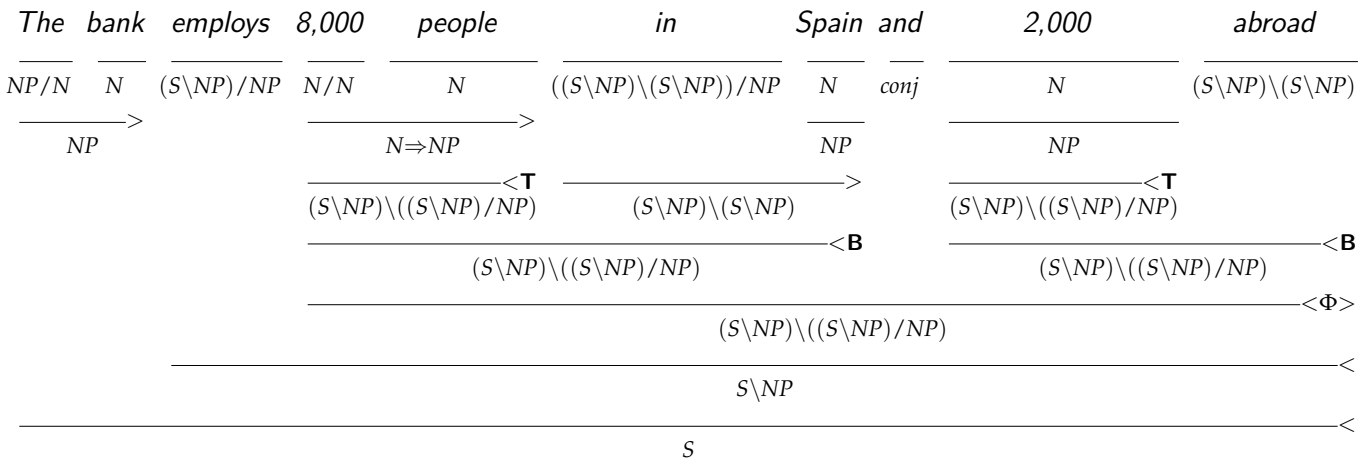
Forward crossed composition

$$X/Y \;\; Y\backslash Z \;\; \Rightarrow \;\; X\backslash Z \quad (>\mathbf{B}_X)$$

Backward crossed composition

$$Y/Z \;\; X\backslash Y \;\; \Rightarrow \;\; X/Z \quad (<\mathbf{B}_X)$$

Steedman (2000) bans the use of forward crossed composition in English as it leads to overgeneration. By applying the same set of type-raising and composition rules to *2,000 abroad*, we can produce equivalent (non-constituent) constructions which can be coordinated together with the coordination rule. The resulting constituent can then consume the verb *employs* to produce the desired final analysis, depicted in Parse 2.6.



Parse 2.6: The CCG analysis of the sentence in Parse 2.1, demonstrating the application, type-raising, composition, and coordination rules.

2.3.3 CCG Dependencies

As complex categories are combined with their arguments, they create a *logical form* representing the semantics of the sentence. Each combinatory rule corresponds to a semantic interpretation which propagates the logical form through the derivation. There are different approaches to representing the semantics, and a commonly used one is the dependencies used in CCGbank (Hockenmaier and Steedman, 2007), the primary corpus of English CCG derivations. Using this mechanism, CCG derivations can generate head-argument dependencies similar to those of Section 2.2.

Dependencies in CCG are created by complex categories. Each argument slot in a complex category is numbered and associated with a corresponding head variable. The dependencies themselves are 5-tuples $\langle h, c, s, f, l \rangle$. In this notation, c is the CCG category which creates the dependency; h is the word to which that category was assigned; s specifies which slot is being filled; and f is the head word of the constituent filling slot s; f will fill the variable that is co-indexed with slot s. Finally, l may contain either $(-)$ or another category, indicating whether the dependency is *local* or *long range*. Long-range dependencies occur when the structures that are related are separated by

$$\textit{The} := (NP_Y/N^1_Y)_{\textit{The}}$$

$$\textit{bank} := N_{\textit{bank}}$$

$$\textit{employs} := ((S_{\textit{employs}}\backslash NP^1_Y)/NP^2_Z)_{\textit{employs}}$$

Figure 2.2: The fully annotated categories for *The*, *bank*, and *employs* in Parse 2.6. Subscript variables Y and Z show the co-indexation mechanism and superscripts indicate slot numbers; combinator application triggers unification of all co-indexed variables on a category.

intervening constituents. If the dependency is long-range, then l contains the CCG category via which the dependency was formed.

CCG dependency creation is driven by the interaction of categories. Unlike in constituency or dependency grammars, where head-argument relationships are usually defined outside the lexicon, the category assigned to a word specifies precisely the dependencies that the word will head, and indirectly specifies the dependencies that the word will be an argument in. Words may head multiple dependencies, or be an argument in multiple dependencies.

Figure 2.2 gives the fully annotated categories used in the C&C parser (discussed further in Chapters 5 to 7) for some of the words from Parse 2.6. *The*, a determiner, requires a noun argument to its right to produce a noun phrase result category. Its fully annotated category is $(NP_Y/N^1_Y)_{\textit{The}}$, indicating that its argument slot 1 is co-indexed with the head of the N argument category that it is seeking. This head will be stored in the variable Y on the category, and is said to be a *filler* of the variable. Upon combination with the N, the result NP is then returned. The co-indexed Y variable indicates that the resulting NP has the same head as the argument N category (*bank*), and that the head word will be available as variable X on that category for further dependencies to use. The variable on the result is denoted as an *active* variable; this is in contrast to *inactive* variables, which have become dormant after combination and are thus no

longer available for dependencies to be formed. The process of passing variables during category combination is termed *unification*, as set and unset variables on the argument are unified through co-indexation with those on the result category.

The transitive verb *employs* requires a subject NP argument and an object NP argument. Its fully annotated category is $((S_{employs}\backslash NP^1_Y)_{employs}/NP^2_Z)_{employs}$. This category indicates that argument slot 1 is associated with variable Y, unified with the variable heading the subject NP, and so when the slot is filled a dependency will be established between *employs* and the word that is assigned to Y. In Parse 2.6, slot 1 of *employs* is filled by *bank*, which is the head of the left NP absorbed by the transitive verb. Slot 2 is associated with variable Z, and this is filled *twice* (by *people* and *2,000*) due to the coordination rule, which creates multiple filler words for a particular variable and thus allows multiple words to fill a particular slot. This process has created dependencies between the words that are similar to the *subject* and *object* relations present in most dependency schemes. However, CCG has given a clear representation for the impact of the coordinated NPs, showing exactly which heads are coordinated, and what syntactic role each plays in the derivation. The parallel structure is also captured in the derivation. The full set of dependencies for the derivation in Parse 2.6 is depicted in Figure 2.3, with the dependencies in blue being those headed by *employs*.

Parse 2.7 depicts the CCG analysis of the ambiguous PP coordination from Parse 2.3. Different lexical categories have been assigned to the words, allowing the prepositions *on* and *under* to be combined using the coordination combinator into a single constituent, which then modifies the verb phrase *opened 8 branches*. The common element of interaction with *employs* is maintained by CCG in Parses 2.6 and 2.7, but the different parallel structure resulting from the coordination is also represented, in contrast to the dependency grammar. The resulting dependencies from the derivation in Parse 2.7 are depicted in Figure 2.4.

$\langle \textit{The}, NP[nb]/N^1, 1, \textit{bank}, - \rangle$

$\langle \textit{employs}, (S[dcl]\backslash NP^1)/NP^2, 1, \textit{bank}, - \rangle$

$\langle \textit{employs}, (S[dcl]\backslash NP^1)/NP^2, 2, \textit{people}, - \rangle$

$\langle \textit{employs}, (S[dcl]\backslash NP^1)/NP^2, 2, \textit{2,000}, - \rangle$

$\langle \textit{8,000}, N/N^1, 1, \textit{people}, - \rangle$

$\langle \textit{in}, ((S\backslash NP)\backslash(S^2\backslash NP))/NP^3, 2, \textit{employs}, - \rangle$

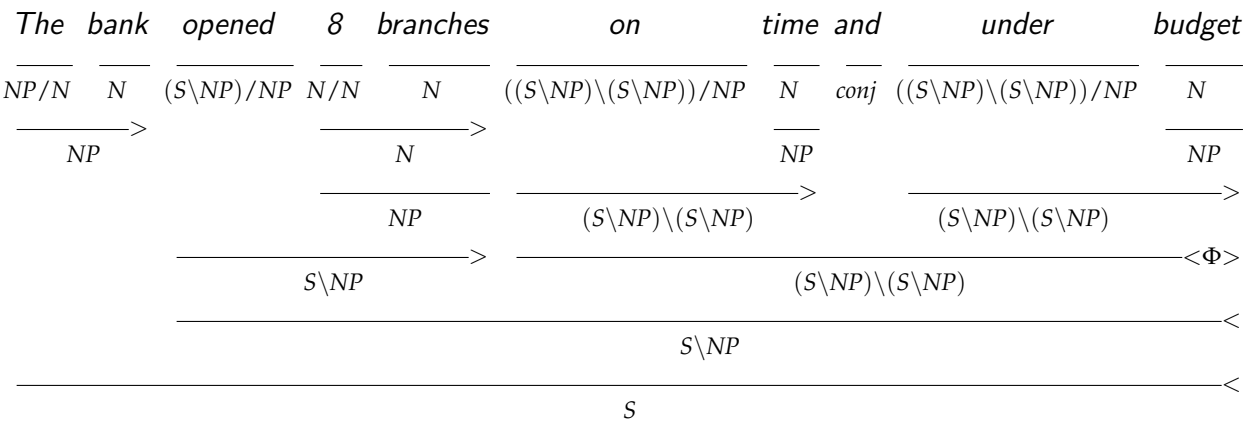
$\langle \textit{in}, ((S\backslash NP)\backslash(S^2\backslash NP))/NP^3, 3, \textit{Spain}, - \rangle$

$\langle \textit{and}, \textit{conj}, 1, \textit{people}, - \rangle$

$\langle \textit{and}, \textit{conj}, 1, \textit{2,000}, - \rangle$

$\langle \textit{abroad}, (S\backslash NP)\backslash(S^2\backslash NP), 2, \textit{employs}, - \rangle$

Figure 2.3: The full set of CCG dependencies generated by the derivation in Parse 2.6.



Parse 2.7: The CCG analysis of the sentence in Parse 2.3. The different underlying coordination is readily identifiable through the different lexical categories assigned to words and the different final derivation.

$$\langle \textit{The}, NP[nb]/N_1, 1, \textit{bank}, - \rangle$$

$$\langle \textit{opened}, (S[dcl]\backslash NP_1)/NP_2, 1, \textit{bank}, - \rangle$$

$$\langle \textit{opened}, (S[dcl]\backslash NP_1)/NP_2, 2, \textit{branches}, - \rangle$$

$$\langle \textit{8}, N/N_1, 1, \textit{branches}, - \rangle$$

$$\langle \textit{on}, ((S\backslash NP_1)\backslash(S_2\backslash NP))/NP_3, 1, \textit{opened}, - \rangle$$

$$\langle \textit{on}, ((S\backslash NP_1)\backslash(S_2\backslash NP))/NP_3, 3, \textit{time}, - \rangle$$

$$\langle \textit{and}, \textit{conj}, 1, \textit{on}, - \rangle$$

$$\langle \textit{and}, \textit{conj}, 1, \textit{under}, - \rangle$$

$$\langle \textit{under}, ((S\backslash NP_1)\backslash(S_2\backslash NP))/NP_3, 3, \textit{budget}, - \rangle$$

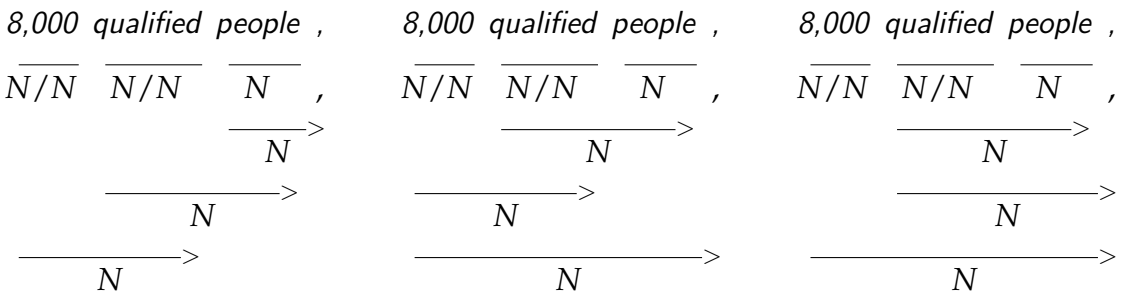
$$\langle \textit{under}, ((S\backslash NP_1)\backslash(S_2\backslash NP))/NP_3, 1, \textit{opened}, - \rangle$$

Figure 2.4: The full set of CCG dependencies generated by the derivation in Parse 2.7.

2.3.4 Ambiguity in CCG

The expressiveness afforded by CCG's combinatory rules leads to what has been termed *spurious ambiguity* (Wittenburg, 1986), where a sentence may have multiple, semantically equivalent derivations. In particular, the type-raising and composition combinators allow the same categories to be combined in different ways by rewriting atomic categories. This functionality is necessary for linguistic phenomena such as the non-constituent coordination of Parse 2.6. However, these combinators allow alternative analyses for forward and backward application, though each derivation still generates the same dependencies and underlying logical form (hence the term 'spurious').

Spurious ambiguity led to criticisms that CCG could not be efficient, as it exponentially increases the potential search space of a parser. Wittenburg (1987) suggested replacing the type-raising and composition combinators with predictive forms that are only used when necessary. Eisner (1996b) noted that this proposal introduced non-



Parse 2.8: Absorption ambiguity in CCG, where all three derivations have a different syntactic structure, but generate identical dependencies.

standard constituents rejected by traditional CCG, and instead described a technique that defined a single *normal-form derivation* for each semantically equivalent analysis (termed an *equivalence class*). The normal-form constraints proposed by Eisner apply to the result categories of composition:

1) No category produced by forward composition may serve as the left child to forward composition or application.

2) No category produced by backward composition rule may serve as the right child to backward composition or application.

Applying these rules during CCG parsing completely eliminates ambiguity when type-raising is not used, and significantly reduces the number of potential derivations when it is used (Clark and Curran, 2004a). They also remove the need for costly semantic equivalence checks during parsing.[1]

Absorption ambiguity in CCG occurs when constituents may be placed in several locations in the derivation without affecting the logical form. That is, the absorption of the constituent generates no dependencies. Punctuation such as commas, brackets, and periods are particularly prone to absorption ambiguity in CCG as the formalism will only assign them a lexical category if they are syntactically important in the sentence (e.g. a comma acting as a coordinator will be assigned the category *conj*). In all other

[1]Hockenmaier and Bisk (2010) point out that the Eisner rules do not preserve all interpretations when the degree of composition is bounded, and present a modified set of constraints satisfying this property.

cases, punctuation is assigned a category based on its POS tag, and will typically be *absorbed* into another constituent using a number of non-standard rules. Parse 2.8 depicts three possible derivations of a short sentence fragment suffixed with a comma, where all three generate identical dependencies.

White and Rajkumar (2008) have shown that absorption ambiguity causes punctuation to overgenerate in a CCG realiser. By assigning lexical categories to punctuation and integrating it more closely into the grammar, they are able to constrain overgeneration and improve realisation coverage and BLEU scores. However, this would not affect all of the ambiguities in Parse 2.8, as they are created by the order of combinator application. Also, within parsing, punctuation is often inconsistently treated in grammars and corpora, and is commonly excluded from parser evaluation (Black et al., 1991; Yamada and Matsumoto, 2003; Buchholz and Marsi, 2006; Clark et al., 2002).

2.4 Corpora and Corpus Conversions

Statistical modelling has engendered wide-coverage, accurate parsers. The most accurate parsers use supervised machine learning techniques to extract a model from an annotated dataset. Unsupervised parsers have been the subject of substantial research over the last decade, but still perform significantly worse than supervised parsers. Thus, the quality of parsing is strongly reliant on the quality of the available annotated data. In this section we briefly introduce constituency, dependency, and CCG corpora and corpus conversion tools that have become widely used for research in parsing, including the Penn Treebank, OntoNotes, LTH and Stanford dependencies, and CCGbank.

2.4.1 Constituency Grammar Corpora

English parsing has been defined for over two decades by one corpus: the Penn Treebank (PTB; Marcus et al., 1993). Released in the early 1990s, the Treebank is a syntacti-

cally annotated corpus of English, and its primary component is the 1.1 million words of newswire from the Wall Street Journal (WSJ). English parsing research has focused heavily on the WSJ data, and the development of the Treebank drove the adoption of statistical parsing using supervised machine learning algorithms.

The Penn Treebank annotation was performed by linguistically-trained annotators correcting and combining fragments from the output of the Fidditch parser (Hindle, 1983). The corpus is distributed in a bracketed-string format representing the constituency trees. An example sentence from the Treebank is presented in Figure 2.5, depicting the sentence in Parse 2.1. POS tags form the preterminals of each node in the tree, with phrasal labels serving as the nonterminals. Co-indexed variables to represent the gapped verb are present, as well as additional function tags indicating the subject (*SBJ*) of the sentence, and that the fragments *Spain* and *abroad* indicate locations (*LOC*).

The exact annotation scheme was motivated by cost-effectiveness, particularly the time expense of correcting the Fidditch output. For example, null elements were retained in the corpus as they were already produced by the parser, and it was relatively simple to verify them. Marcus et al. (1993) notes that this decision had already enabled early work on recovering predicate-argument dependencies from the Treebank, and the presence of null elements would be crucial for the later conversions of the Penn Treebank to dependency and linguistically-motivated grammar formalisms. On the other hand, the distinction between complements and adjuncts could not be made consistently by annotators, and the effort required for this slowed down the annotation process considerably.

Noun phrases were also deficiently bracketed due to the unacceptable cost of inserting more fine-grained brackets than those provided by the parser. Thus, the original Treebank assumes that all NPs have a flat structure with all constituents at the same level, though this is clearly not always the case in language, e.g. *((Air Force) contracts)* and *((crude oil) prices)*.

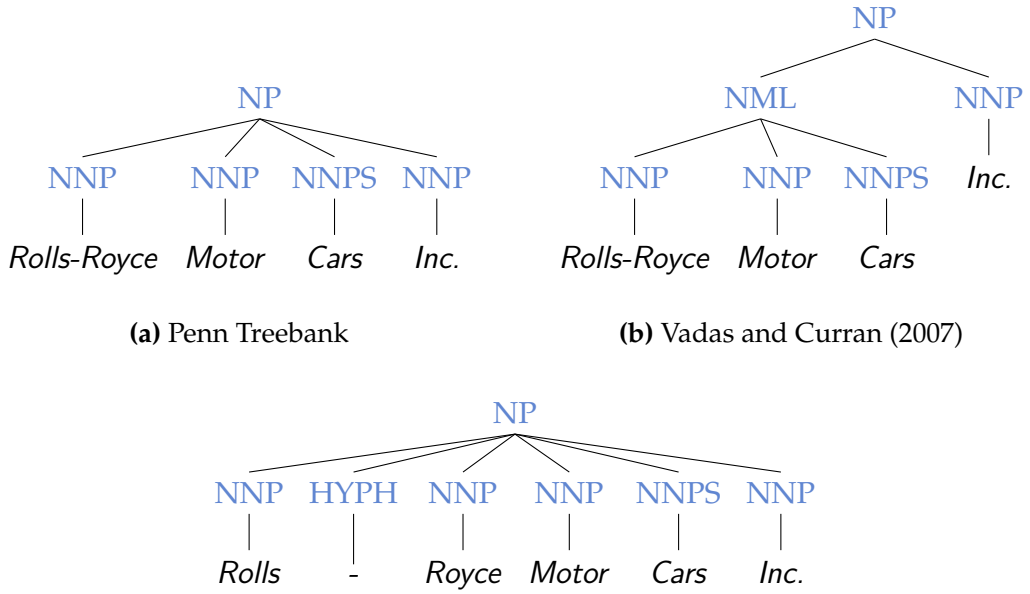
```
( (S
    (NP-SBJ (DT The) (NN bank) )
    (VP
      (VP (VBZ employs)
        (NP-1 (CD 8,000) (NNS people) )
        (PP-LOC-2 (IN in)
          (NP (NNP Spain) )))
      (CC and)
      (VP
        (NP=1 (CD 2,000) )
        (ADVP-LOC=2 (RB abroad) )))
    (. .) ))
```

Figure 2.5: The Penn Treebank bracketed string representation of the tree in Parse 2.1.

The Treebank was unmatched in scale and detail at the time of its release, and provided the missing resource required for statistical parsing. It quickly became a de-facto standard, and corpora for many other grammars, including dependency schemes (Johansson and Nugues, 2007; de Marneffe and Manning, 2008a), CCG (Hockenmaier and Steedman, 2007), Lexicalised Tree-Adjoining Grammar (Shen et al., 2008), and Head-driven Phrase Structured Grammar (Miyao et al., 2004) have been automatically and semi-automatically derived from the Treebank.

There has been considerable attention paid to the WSJ data since the initial release of the Penn Treebank, with several augmentations to the corpus annotation. Vadas and Curran (2007) added complete noun phrase bracketing to the treebank, enriching the internal NP structure and allowing parsers built on the corpus to model the syntactic and semantic structure inside. As we will discuss later, this work has proved important for converting the Treebank data to different grammar formalisms. Meanwhile, different types of annotation have been applied on top of the syntactic trees in the corpus, including semantic roles (Palmer et al., 2005) and nominals (Meyers et al., 2004).

The OntoNotes project has sought to create a holistic, semantically enriched corpus that combines syntactic annotation with shallow semantics (Hovy et al., 2006; Weischedel et al., 2011). The Penn Treebank WSJ data is included in the OntoNotes



(a) Penn Treebank

(b) Vadas and Curran (2007)

(c) OntoNotes 4

Figure 2.6: The noun phrase structure assigned under three different annotations of the Penn Treebank data. Each phrase is headed by *Inc.*

release, though the original annotation has been changed in several ways, and some sentences have been excluded. The raw newswire sentences were retokenised with slightly different rules, most notably splitting hyphenated words into separate tokens (Warner et al., 2004). Additional HYPH and AFX POS tags were added to account for the new tokens introduced from splitting hyphens. The syntactic annotation guidelines were also updated to improve the detail provided in NPs. However, these changes are not always compatible with those of Vadas and Curran (2007) due to the tokenisation changes. Figure 2.6 depicts the differences between the original Penn Treebank, the Vadas and Curran (2007) version, and the OntoNotes 4 release for a noun phrase containing a hyphenated proper noun.

Parsers have been shown to display domain dependence, where they perform much better on text taken from the same domain as their training data compared to text from outside that domain (Sekine, 1997; Gildea, 2001). An ongoing challenge is to improve out-of-domain performance, particularly on web text, which presents its own unique

challenges. Unlike heavily-edited newswire, web text sourced from blogs, discussion forums, and QA sites often display inconsistent capitalisation and punctuation usage, spelling and grammatical mistakes, and heavy use of jargon and slang. Syntactic constructions such as questions, imperatives, lists, and fragments are also much more common in web text than newswire.

The English Web Treebank (EWT) is a collection of 16,624 sentences of web text annotated with similar syntactic structure to the Penn Treebank (Bies et al., 2012). It is the largest syntactically annotated corpus of web text currently available, and it was used for the first shared task of the Syntactic Analysis of Non-Canonical Language workshop (SANCL; Petrov and McDonald, 2012). The data in the corpus is split roughly evenly across five domains:

- question-answers taken from Yahoo! Answers, a community-driven QA site;
- newsgroup discussion threads pertaining to a variety of topics and interest area;
- online reviews of businesses and services taken from various Google websites;
- email messages sent by employees of the Enron Corporation, taken from the Enronsent Corpus (Styler, 2011);
- weblog posts and articles.

The SANCL 2012 Shared Task provided development and test splits of the English Web Treebank by partitioning the sentences in each domain in half. The first half of each domain is used as development data, and the second half as test data.[2]

2.4.2 Dependency Corpora

Dependency treebanks are typically developed in two ways. Some treebanks are manually constructed over raw data, and others are semi-manually produced by correcting or adapting the output of an existing parser. Other dependency treebanks are automatically induced from existing corpora, most notably the Penn Treebank and its variations.

[2]Slav Petrov, p.c.

The automatic conversion process is typically developed following some amount of manual inspection and experimentation with the existing corpus.

The Prague Dependency Treebank (PDT) contains over 38,000 sentences of Czech text, sourced from newspapers and magazines (Hajič, 1998). Inspired by the Penn Treebank, it is manually annotated with morphological information, dependency structures, and linguistic meanings. The PDT set a de-facto standard for dependency treebank representation, and has seen continuing updates since its introduction – the most recent of which is the PDT 3.0 (Bejček et al., 2013). The PDT also set a precedent in providing a large annotated corpus in a language other than English – a forerunner to the widespread recent work in multilingual dependency parsing.

The PARC 700 Dependency Bank (*DepBank*) consists of 700 randomly chosen sentences from section 23 of the original Penn Treebank. Each sentence was parsed using a Lexical-Functional Grammar parser before manual correction and adjustment to produce a gold-standard dependency output (King et al., 2003). The grammar used in DepBank is broadly similar to the *grammatical relations* (GRs) scheme of Briscoe (2006), used in the RASP unlexicalised grammar parser (Briscoe et al., 2006). In both the DepBank and GR schemes, dependencies are binary-labeled relations between head and argument. However, DepBank includes more detailed feature-based information, such as noun plurality, number type, and form (Briscoe and Carroll, 2006). Both grammars produce dependency graphs without an explicit root, and each word may have multiple heads. The GR hierarchy is depicted in Figure 2.7.

Early research on statistical parsing with the Penn Treebank identified phrase head words as an important feature. Both Magerman (1994) and Collins (1999) provide percolation "head-finding" rules for the Treebank to heuristically locate heads based on the labels assigned to the phrase and the constituents it contained. Recursively extracting the head-modifier relationships from each phrase using these heuristics gives a set of dependencies spanning the sentence (as described in Section 2.2). These unlabeled dependencies, extracted over the Penn Treebank, formed a projective, automatically-

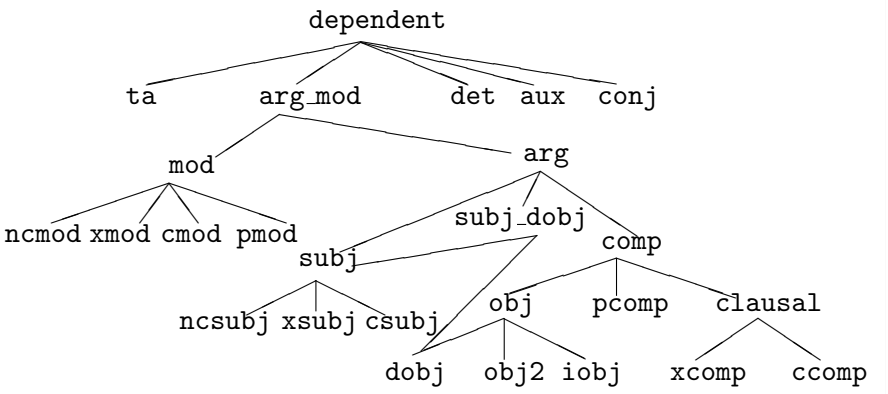


Figure 2.7: The grammatical relations hierarchy, reproduced from Briscoe (2006).

extractable dataset that was used in early dependency parsers, such as Eisner (1996c). Yamada and Matsumoto (2003) provided a set of heuristics extending Magerman (1994). Nivre (2006) reimplemented their algorithm, incorporating heuristics to infer labels for each of the dependency arcs. The dependency scheme came to be known after the name of the Nivre (2006) tool to produce it, Penn2Malt.

More recent dependency schemes have sought to improve on the early heuristic head-finding rules in a number of ways. The LTH scheme focused on producing dependencies with improved linguistic fidelity whilst maintaining the constraint of a single head per word. It took advantage of the traces, co-indexation, and null elements in the Treebank that were essentially ignored by earlier converters (Johansson and Nugues, 2007). The scheme also utilised the improved Treebank NP annotation of Vadas and Curran (2007) to more accurately model NP internal structure. The resulting scheme has a richer set of labels than Penn2Malt, and can handle long-distance phenomena such as the gapping example presented in Section 2.2, *wh*-movement, and expletives. Johansson and Nugues (2007) note that raising and control are challenging to model with only one head per word. Due to the increased syntactic detail, the scheme can produce trees which are non-projective. However, the majority of analyses in English are projective, and prior work with these dependencies in English has used projective parsing algorithms to good effect (Bansal and Klein, 2011).

1	*The*	DT	2	*det*
2	*bank*	NN	3	*nsubj*
3	*employs*	VBZ	0	*root*
4	*8,000*	CD	5	*num*
5	*people*	NNS	3	*dobj*
6	*in*	IN	3	*prep*
7	*Spain*	NNP	6	*pobj*
8	*and*	CC	3	*cc*
9	*2,000*	CD	3	*conj*
10	*abroad*	RB	9	*advmod*

Figure 2.8: A compressed tabular representation of the Stanford dependency tree in Parse 2.2, based on the CoNLL format. Each word of the sentence is given an index, with the head of each word indicated by the appropriate index. By convention, the root word is headed by index 0.

LTH has seen widespread use in recent work, including the 2007, 2008, and 2009 CoNLL Shared Tasks on dependency parsing. These tasks spurred an increased research focus on multilingual parsing (Nivre et al., 2007; Surdeanu et al., 2008; Hajič et al., 2009). The grammar has also been shown to improve the performance of downstream tasks due to its richer linguistic information (Elming et al., 2013). The pennconverter tool[3] accepts bracketed constituency trees for sentences in Penn Treebank-like formats, and produces LTH dependencies for those sentences in the tabular CoNLL format, depicted in Figure 2.8.

The Stanford dependency scheme (SD; de Marneffe and Manning, 2008a) has also become widely popular for dependency parsing work (McDonald et al., 2013). Stanford dependencies have proved to be useful for a wide variety of tasks, including parser evaluation (Cer et al., 2010; Nivre et al., 2010), recognising textual entailments (Dagan et al., 2009), and biomedical natural language processing (Pyysalo et al., 2007). Stanford dependency corpora have been produced for a variety of different languages, culmi-

[3] http://nlp.cs.lth.se/software/treebank_converter/

nating in the recent development of a Universal Dependency Scheme based upon it (McDonald et al., 2013).

Stanford dependencies were designed with the intention of producing semantically contentful relations useful in downstream tasks. For example, the distinction between complements and adjuncts is syntactically important, but is less useful in relation extraction tasks. In contrast, NP internal relationships are much more critical, and so Stanford dependencies largely ignore the former and include many labels for the latter (de Marneffe and Manning, 2008b).

Stanford dependencies can be extracted in two main flavours. The *basic* variant produces projective trees, with a single word per each head. The *collapsed* variant removes intervening function words from the representation. For instance, coordinators are changed into relations under the collapsed variant, having the effect directly linking coordinated works and removing the less semantically useful conjunction dependency. The Stanford parser[4] (de Marneffe et al., 2006) provides the canonical converter from Penn Treebank-style constituency trees to Stanford dependencies in basic and collapsed forms. Unlike LTH, the Vadas and Curran (2007) enriched NP brackets is not mandated, and the conversion does not make use of the trace information in the Treebank.

2.4.3 CCG Corpora

CCGbank (Hockenmaier and Steedman, 2007) is the primary corpus used for English parsing with CCG. It was derived from the Penn Treebank WSJ data through a semi-automated process (Hockenmaier, 2003b). Similar CCG treebanks have been induced for German (Hockenmaier, 2006) and Chinese (Tse and Curran, 2010).

Hockenmaier (2003a) describes the process used to transform the Penn Treebank into CCGbank. The process converted 99.44% of the WSJ sentences, with the remainder being fragments, failed derivations due to the adjunct-finding heuristics used in the procedure, or sentences exhibiting linguistic phenomena that were not covered by

[4] http://nlp.stanford.edu/software/lex-parser.shtml

```
(<T S[dcl] 0 2>
  (<T S[dcl] 1 2>
    (<T NP 1 2>
      (<L NP[nb]/N DT The>)
      (<L N NN bank>) )
    (<T S[dcl]\NP 0 2>
      (<L (S[dcl]\NP)/NP VBZ employs>)
      (<T (S\NP)\((S\NP)/NP) 1 2>
        (<T (S\NP)\((S\NP)/NP) 0 2>
          (<T (S\NP)\((S\NP)/NP) 0 1>
            (<T NP 0 1>
              (<T N 1 2>
                (<L N/N CD 8,000>)
                (<L N NNS people>) ) ) )
        (<T (S\NP)\(S\NP) 0 2>
          (<L ((S\NP)\(S\NP))/NP IN in>)
          (<T NP 0 1>
            (<L N NNP Spain>) ) ) )
      (<T (S\NP)\((S\NP)/NP)[conj] 1 2>
        (<L conj CC and>)
        (<T (S\NP)\((S\NP)/NP) 0 2>
          (<T (S\NP)\((S\NP)/NP) 0 1>
            (<T NP 0 1>
              (<L N CD 2,000>) ) )
          (<L (S\NP)\(S\NP) RB abroad>) ) ) ) ) )
  (<L . . . . .>) )
```

Figure 2.9: The CCGbank representation of the derivation in Parse 2.6, which is much deeper than the PTB representation in Figure 2.5 due to the binary-branching. factor. Internal nodes (<T) specify category, headedness (0 = left child head), and number of children.

the algorithm. CCGbank contains 48,934 sentences of CCG normal-form derivations, represented as binary-branching trees. Leaves in the tree specify the words of the sentence with their POS tags and categories. Internal nodes represent the results of combinations, and specify the result category, headedness (with a 0 indicating the node is headed by its left child, and 1 indicating the right child), and the number of children. Figure 2.9 gives the CCGbank representation of the CCG derivation in Parse 2.6.

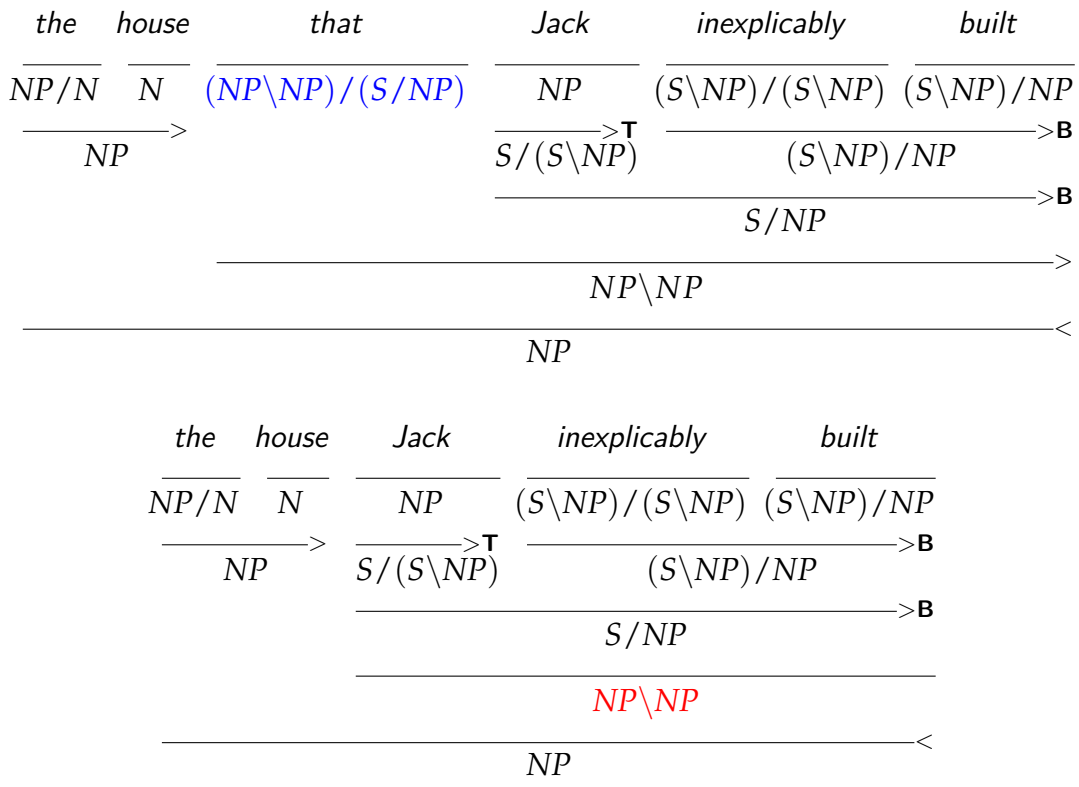
CCGbank introduces *features* to atomic categories, which convey additional linguistic information. For example, declarative clauses are labelled $S[dcl]$, while passive

clauses are labelled $S[pss]$. Noun phrases beginning with a determiner are labelled $NP[nb]$ to indicate that they are non-bare.

Parallel to the normal-form derivations, CCGbank also lists the CCG dependencies generated by each derivation, and the co-indexation and variable-passing behaviour for each category required to create dependencies. These gold dependencies are the standard data used for evaluating CCG parsers, and the co-indexation information is crucial for developing a CCG parser itself. One deficiency of the co-indexation mechanism is that it assumes all words bearing a category yield the same dependencies; that is, co-indexation information is unique given the category. Hockenmaier and Steedman (2007) notes that this is most obviously incorrect in the case of control verbs, such as *promise* and *persuade*, which are annotated with the same category, but should be co-indexed differently and produce different dependencies.

A number of non-standard rules and adaptations of CCG's combinators were introduced in CCGbank to handle idiosyncrasies in the Penn Treebank annotation, as well as constructions that cannot be elegantly analysed by CCG. The most prevalent amongst these are a number of unary type-changing rules that allow several kinds of VP constituents to become modifiers of NPs and VPs. These constructions are called *clausal adjuncts*, and are challenging to encode in CCG as the clausal adjunct exhibits different syntactic behaviour to the verb phrase, necessitating a different category. Some of the unary type-changing rules used in CCGbank are listed below:

$$
\begin{aligned}
N &\Rightarrow NP \\
S[pss]\backslash NP &\Rightarrow NP\backslash NP \\
S[adj]\backslash NP &\Rightarrow NP\backslash NP \\
S[ng]\backslash NP &\Rightarrow NP\backslash NP \\
S[ng]\backslash NP &\Rightarrow (S\backslash NP)\backslash(S\backslash NP) \\
S[dcl]/NP &\Rightarrow NP\backslash NP
\end{aligned}
$$



Parse 2.9: A CCG analysis of a relative clause and a reduced relative clause excluding the relativiser *that*. A unary rule is necessary to perform the syntactic function of *that*, otherwise the canonical categories assigned to *built* and its modifier *inexplicably* must be changed.

These rules are included to prevent excessive category proliferation, as all verbs which could be used as modifiers would require alternative categories when they appeared in the modifier role. These additional categories would then propagate through categories which modify those verbs. This problem illustrates a weakness of CCG, in that it is difficult to encode syntactic behaviour without a corresponding lexical unit. Parse 2.9 depicts a reduced relative clause where the unary rule performs the role of the omitted relativiser, *that*. This allows the same categories as the non-reduced clause to be used.

Containing the category set size using unary rules decreases the ambiguity of the lexicon, and increases generalisation. However, these unary type-changing rules bring syntactic behaviour out of the lexicon and into the grammar, interfering with a core aim of the formalism. They also affect the dependency creation process, as the result

category of the unary rule must mediate any dependencies projected from within the adjunct clause — dependencies which may not be consistent across all applications of the rule. The impact of these inconsistencies has not been well studied in CCG parsing.

Honnibal (2010) proposed an alternative to unary type-changing rules in the form of *hat categories*, which incorporate the type-changing rules as a feature of the verb category itself. The desired modifier category is modeled as a different 'hat' which the verb can wear if required. However, using hat categories increases the size of the lexicon, and reduces parsing accuracy compared to using the unary rules. We do not make use of hat categories in this thesis.

Rather than the ternary coordination rule presented in Section 2.3.2, CCGbank represents coordination with a set of binary rules as follows:

$$\begin{aligned} conj\ X &\Rightarrow X[conj] \\ ,\ X &\Rightarrow X[conj] \\ X\ X[conj] &\Rightarrow X \end{aligned}$$

These rules can be seen in the CCGbank derivation of Figure 2.9, where the ternary rule presented in Parse 2.6 has been binarised. This has the advantage of ensuring that all trees in CCGbank are at most binary-branching, rather than including ternary trees where coordination is present.

As CCGbank was induced from the original Penn Treebank, it inherits some problems from the source corpus, such as the lack of NP structure. While the Penn Treebank annotates NPs as flat constructions, CCG is binary branching, and so the conversion algorithm simply assumed that all NPs were exclusively right-branching. This creates incorrect analyses for left-branching noun phrases and has unwelcome side-effects when combined with coordination within the phrase. The following nonstandard *conj* absorption rule is included in CCGbank to handle these cases:

$$conj\ N \Rightarrow N$$

The improvements in NP annotation from Vadas and Curran (2007)'s reannotated version of the Penn Treebank were transferred to CCGbank by Vadas and Curran (2008), though we do not make use of these changes in this thesis.

2.5 Parser Evaluation

Evaluation is typically performed by running a parser over a dataset with known correct annotations, and comparing the parser output against the annotated gold standard. The reliance on linguist-annotated corpora has led to a standardisation in methodology and evaluation in parsing. In particular, the PTB WSJ has become the de-facto standard corpus for training and evaluating supervised constituency parsers. In English, the corpus is divided into 25 sections, numbered from 00 to 24; historically, sections 02-21 have been used as parser training data, sections 00 and 22 for parser development, and section 23 for parser evaluation.

Dependency parsers and CCG parsers are commonly evaluated on converted Penn Treebank data. However, the precise mechanics of the evaluation process are very different for these parsers. In this section, we describe how parsers for different grammars are evaluated, and discuss a variety of issues with the current evaluation practices.

2.5.1 Constituency Evaluation

The standard evaluation metric for constituency parsers are the PARSEVAL scores (Black et al., 1991) over a held-out section of the PTB. PARSEVAL measures the *constituent accuracy* of each individual sentence — how similar the constituents in the analysis produced by the parser are to the gold standard. However, as erroneous constituents in the parser output may not necessarily correspond directly to the constituents in the gold standard, constituent accuracy alone is not a meaningful result. Instead, precision and recall metrics are calculated to identify the percentage of correct constituents in the

parser output, and the percentage of expected constituents that appear in the parser output. The harmonic mean of these gives the PARSEVAL F-score for the sentence.

The PARSEVAL metric is extremely sensitive as it is calculated on a per-bracket per-sentence basis. A change of a few tenths of a percent is usually sufficient to be statistically significant over the standard evaluation dataset. However, it is also extremely coarse; Lin (1995) notes that a single misplaced constituent can trigger multiple precision and recall errors, and all constituents are given equal weighting, rather than assigning semantically crucial constituents more importance. Long-distance dependencies are also difficult to measure, as they rely on traces or co-indexation which is commonly not present in constituency output.

The nature of the PTB data typically used with PARSEVAL creates further drawbacks. Manning and Carpenter (2000) point out that parses in the Treebank are typically quite flat, meaning that there are relatively few internal constituents. The lack of internal NP structure is a widespread example where the Treebank does not provide detailed constituencies. Additionally, the way in which the often ambiguous PP constituent is annotated in the Treebank allows for mistaken attachments to go lightly punished by PARSEVAL.

Despite these flaws, calculating the PARSEVAL score aggregated over all sentences in the PTB section 23 allows a direct comparison with over twenty years of results in constituency parsing research (Magerman, 1995; Collins, 1996, 1997; Charniak, 2000; Charniak and Johnson, 2005; McClosky et al., 2006; Petrov and Klein, 2007).

2.5.2 Dependency Evaluation

Dependency parsers have been traditionally evaluated directly on the accuracy of head attachment decisions, or *attachment score*. *Unlabeled attachment scores* (UAS) evaluate only whether the head and argument of a dependency is correct, and *labeled attachment scores* (LAS) also require the assigned label to the dependency to be correct. Unlike constituency parsers, dependency grammars that assign a single head to each word

can be meaningfully evaluated on accuracy alone as there is a direct correspondence between the parser-produced output and the gold standard. Under this, every word contributes equally to the overall accuracy.

Unlabeled attachment scores have been widely used to evaluate parsers on several dependency schemes, including Penn2Malt dependencies (Yamada and Matsumoto, 2003; McDonald et al., 2005a; Nivre, 2006), LTH dependencies (Bansal and Klein, 2011; Pitler, 2012a), and Stanford dependencies (de Marneffe et al., 2006; Cer et al., 2010).

Lin (1995) and Carroll et al. (1998) argue for a dependency-based evaluation regime for all parsers, citing the weaknesses of the PARSEVAL constituency metric, and advocating the cross-formalism potential of a relatively simple, but semantically meaningful dependency scheme. They propose precision, recall, and F-score metrics over the parser-recovered dependencies as compared to the gold standard dependencies. The scheme generalises over each parser's internal representation, and allows for long-distance dependencies to be evaluated. The disadvantage of this cross-parser scheme is the requirement for parsers to convert their native output to the evaluation scheme, potentially introducing errors and artificial performance limits.

Rimell et al. (2009) argue that a single combined metric such as PARSEVAL F-score or UAS offers a skewed view of parser performance. Parsers that perform similarly on an aggregated metric may perform very differently on rare constructions as this difference is masked by their low frequency. They propose a cross-formalism evaluation based on the recovered of *unbounded dependencies*, where any number of intervening constituents may be placed between the head and argument of a dependency. These dependencies are important for building the complete predicate-argument structure of a sentence in downstream tasks, and are difficult to recover using shallow parsing methods.

Rimell et al. build a Stanford dependency corpus of 700 sentences, specifically chosen due to the presence of unbounded dependencies. Most parsers did quite poorly on the evaluation, illustrating the difficulty of recovering unbounded dependencies.

However, a key challenge was extracting the unbounded dependencies from parser output, due to the different dependency representation conventions across each parser.

2.5.3 CCG Evaluation

CCG combines both a constituency- and dependency-type analysis in its output. However, both the PARSEVAL and head attachment metrics are of little use in evaluating CCG parsers. As the grammar is binary branching, CCG trees are deeper and contain more constituents (i.e. brackets) than most constituency grammars – particularly the PTB. Additionally, many different derivations may generate the same underlying dependencies, rendering them semantically equivalent. Such equivalent analyses should be scored identically, rather than being penalised for differing from the gold-standard derivation. While this implies a dependency-based evaluation for CCG, such an evaluation cannot be purely based on attachment accuracy, as each word in a sentence is not restricted to having only one head.

Instead, the standard evaluation for CCG parsers calculates precision, recall, and F-score over the dependencies returned by the parser, similar to Lin (1995) and Carroll et al. (1998). The final evaluation for CCG parsers is typically undertaken over CCGbank section 23 Clark et al. (2002).

2.6 Summary

In this chapter, we have briefly covered the grammars, corpora, and evaluation procedures used in this thesis. We have contrasted the properties of a number of grammar formalisms, described how they are created, and how the data in each formalism can be used for training and evaluating parsers.

We now focus on English dependency parsing, and develop a comprehensive classification of errors made by parsers based on applying constraints during parsing.

3 Constraint-based Error Analysis for Dependency Parsing

In this chapter, we implement dependency-level constraints in graph-based and transition-based dependency parsers, and use the constraints to perform an in-depth exploration of how the parsers respond when forced to generate specific arcs. By grouping dependencies into error classes, we determine the relative *constrained* performance of the parser on each error class on newswire and web text, as well as the *cascading* influence of each class on the remainder of the sentence. This abstracts over the raw attachment scores, and provides a deeper understanding of the comparative performance of the parsers in- and out-of-domain, identifying the highest impact areas to target for potential improvement.

We begin by building upon the background in the previous chapter, sketching a brief history of parsing using the Penn Treebank, and introducing the widely-used graph-based and transition-based approaches to dependency parsing. We motivate the implementation of dependency-level constraints on individual arcs in each parser, describing how we apply constraints to the parsers to force them to produce certain arcs, whilst otherwise continuing with their decoding process. We group dependencies into high-level error classes, and use them to perform an in-depth error analysis of the parsers, comparing their performance across domains.

Dependency parsers are evaluated by measuring the word-level attachment accuracy of heads and labels. Whilst this is easily interpreted and comparable across

systems, it does not provide insights into the types of errors made by the parser, why they are being made, or what cascading impact they have. For example, misidentifying the head of a modifier may only introduce a single attachment error, while misplacing the root of a sentence will create substantially more errors elsewhere. Erroneous arcs can also interfere with tree constraints, impeding the selection of correct arcs in both projective- and non-projective dependency parsing.

Kummerfeld et al. (2012) abstracted away from individual bracket errors in constituency parsing and identified broader error classes spanning across multiple brackets (such as clause attachment and modifier attachment). These error classes are grouped with tree repair operations, allowing parser output to be analysed by repeatedly identifying the error class and repair operation that fixes the most incorrect brackets. Implemented as a post-processing script, the algorithm assumes that the remainder of the parse is static as each error class is repaired. However, it is unclear whether the parser will apply the repair operation in its entirety, or if it will introduce other changes in response to the repairs.

We develop an evaluation procedure to evaluate the influence of each error class in dependency parsing without making assumptions about how the parser will behave. We define error classes based on dependency labels, and use the dependencies in each class as *arc constraints* specifying the correct head and label for particular words in each sentence. We adapt parsers to apply these arc constraints, enforcing correct attachments for those words, whilst otherwise proceeding with decoding under its grammar and model. By evaluating performance with and without constraints, we can directly observe how each constraint set affects the parser, and identify the cascading impact of repairing those errors.

We implement our procedure for the graph-based MSTParser (McDonald and Pereira, 2006) and the transition-based ZPar (Zhang and Clark, 2011b), two widely-used and representative dependency parsers. We evaluate over basic Stanford dependencies generated over the OntoNotes 4.0 release of the WSJ Penn Treebank data. We group

Stanford labels into error classes, and test constraints on words bearing each label in turn. Our results show that erroneously attaching NPs, PPs, modifiers, and punctuation have the largest overall impact on UAS. Of those, NPs and punctuation have the most substantial cascading impact, indicating that these errors have the most effect on the remainder of the parse. Enforcing correct punctuation arcs has a particularly large impact on accuracy, even though most evaluation scripts ignore punctuation. We find that punctuation arcs are commonly misplaced by large distances in the final parse, crossing over and forcing other arcs to be incorrect in the process.

3.1 A Brief History of Computational Parsing

The earliest parsers used a *rule-based* approach. Linguists would curate a grammar (typically using context-free productions) describing how sentences could be legally formed in a language. This grammar would then be encoded in software and applied to natural language directly. Some examples of such systems include the IBM Computer Manuals constituency parser of Black et al. (1993), and the dependency-like Link Grammar parser of Sleator and Temperley (1991).

Unfortunately, manual grammar curation is labour-intensive, and expert linguistic knowledge is required to create the grammar and maintain it as the language evolves. Resolving ambiguity is difficult in rule-based grammars, and even the most complete grammar will grow obsolete over time.

The deficiencies of rule-based parsing influenced a move to statistical parsing, which continues to dominate the field today. The role of the linguist has shifted from defining rules for parsing to creating the annotated resources necessary for training parsers. Annotated data has the advantage of being applicable to many different parsers; the Penn Treebank in particular has convincingly demonstrated the utility of a large syntactically-annotated dataset for multiple grammar formalisms. Standardising

constituency parser evaluation on the PARSEVAL metric over WSJ section 23 also contributed to an intense interest in statistical parsing.

Magerman (1995) describes SPATTER, the first parser trained on the WSJ data. SPATTER used decision tree models predicated on word clustering to construct a constituent tree for each sentence. Statistics for the word clustering were gathered from the WSJ. SPATTER outperformed all existing rule-based parsers on the PARSEVAL metrics, including some that had been under development for a decade. This result convincingly demonstrated that statistical parsing using large treebanks for training data was more viable and robust than grammar engineering.

Magerman's work opened a new era of research work on statistical parsing, alongside renewed development of large annotated corpora across different languages. Collins (1996) and Collins (1997) describe parsing models that decompose the construction of tree constituents into a sequence of steps based on generating the head child first, followed by each other child constituent outward from the head. The generation of each child is assumed to be independent and is conditioned on all previously generated children. This approach is an extension of the process of parsing probabilistic context-free grammars, where probabilities for the model are estimated from the PTB. The parser became known as the Collins parser (Model 1) and it outperformed SPATTER on PARSEVAL. Continued enhancements aimed at capturing more detailed linguistic phenomena such as subcategorisation and *wh*-movement resulted in Models 2 and 3 of the parser (Collins, 1999), which further improved performance. The Collins parser (particularly Model 2) has been widely-used and its generative head-driven parsing process is particularly influential in the field.

Eisner (1996a,c) proposed probabilistic dependency parsing algorithms that were trained and evaluated on the Penn Treebank, achieving comparable accuracies to the contemporary work on statistical constituency parsing. Eisner (2000) further generalised these models and many other dependency formulations as *bilexical grammars* with efficient parsing algorithms. Eisner's work, particularly the seminal parsing algo-

rithm now known as *Eisner's algorithm* (discussed further in Section 3.2.1.1), was highly influential in the development of statistical dependency parsing.

Charniak (1997) describes a probabilistic constituency parser that improved over SPATTER and the early Collins parser on PARSEVAL. Charniak's key improvement was the extraction of a context-free grammar as well as probabilities from the WSJ, allowing a finer level of control over the generated structures. Like the Collins parser, the Charniak parser underwent iterative improvement as new techniques emerged in parsing. Charniak (2000) added a maximum-entropy inspired model (influenced by Ratnaparkhi (1997)'s work on linear-time parsing) and Charniak and Johnson (2005) proposed a reranking system that reordered the top n-best parses for each sentence with a secondary model. The Charniak parser is widely-used and accurate; McClosky et al. (2006)'s work on reranking and self-training for the Charniak parser make it the current state-of-the-art in English constituency parsing with a PARSEVAL labelled F-score of 92.1%.

Klein and Manning (2003b) describe the Stanford unlexicalised parser, which outperformed the early lexicalised parsers of Magerman (1995) and Collins (1996), despite encoding no word-specific information. They illustrated how simple, linguistically-motivated state-splits could capture structural regularities in the PTB, avoiding the sparseness of lexical and bilexical probabilities. Klein and Manning (2003c) use factorisation to combine the unlexicalised model with a lexicalised dependency parser, allowing the structural and predicate-argument features to be scored independently before being combined into an overall probability for each parse. An additional advantage of the resulting Stanford factored parser is that it permits efficient exact inference using the A* search algorithm, as the simpler sub-models can be used to prune the search space of the overall parser.

Petrov et al. (2006) extend the manual state-splitting annotations of Klein and Manning (2003b) into an automatic process, developing a unlexicalised split-merge parser known as the Berkeley parser. The system begins with a coarse X-bar grammar, and

repeatedly splits, re-trains, and merges the symbols over the PTB using the Expectation-Maximisation algorithm. The iterative refinements and compactions result in a parser of comparable performance to the most accurate lexicalised systems. Petrov and Klein (2007) further improve inference in the Berkeley parser, experimenting with reranking and dynamic programming algorithms in place of Viterbi decoding.

A particular interest in multilingual constituency and dependency parsing developed as corpora in multiple languages became available. Collins et al. (1999) describes a dependency parser for Czech, based on the Prague Dependency Treebank (Hajič, 1998). Adapting the model of Collins (1997), the parser converted the dependency structures in the Treebank to linguistic trees internally before extracting dependencies for evaluation. Influenced by this work, Collins' models were widely applied to constituency parsing in many languages, including Chinese (Bikel, 2002), German (Dubey and Keller, 2003), French (Arun and Keller, 2005), and Spanish (Cowan and Collins, 2005). Meanwhile, dependency parsing models were also developed for many languages as corpora became available, including Japanese (Kudo and Matsumoto, 2000), Swedish (Nivre et al., 2004), Bulgarian (Marinov and Nivre, 2005), Chinese (Cheng et al., 2005), and Danish (McDonald and Pereira, 2006). Recent CoNLL Shared Tasks on Multilingual Dependency Parsing (Buchholz and Marsi, 2006; Nivre et al., 2007; Hajič et al., 2009) have led to the creation of dependency corpora with consistent formats, as well as a widespread focus on the challenges of dependency parsing in different languages.

We will discuss CCG parsing in more detail in Chapters 5 to 7. In the next section, we will discuss dependency parsing algorithms in more detail, focusing on the widely-used graph-based and transition-based approaches.

3.2 Dependency Parsing

In this thesis, we work with a graph-based parser (MSTParser), and a transition-based parser (ZPar). These two contrasting systems represent the most common approaches to dependency parsing in the recent literature. In this section we will briefly describe both parsing methodologies and the decoding algorithms that underpin them.

3.2.1 MSTParser: Graph-based Dependency Parsing

Graph-based techniques cast dependency parsing as the assembly of a well-formed dependency tree from smaller tree fragments. The parsing task is directly factored over the smaller fragments, which are typically the dependency arcs of the tree. *First-order* techniques assemble the tree by repeatedly combining the best-scoring individual arcs according to the parser model. The total score for the tree is usually the sum of the scores of all individual arcs used to construct the tree. *Second-order* and *third-order* techniques additionally factor over pairs and triples of arcs respectively, but require more complex decoding algorithms to maintain acceptable time complexity.

MSTParser (McDonald et al., 2005a,b; McDonald and Pereira, 2006) is a popular graph-based dependency parser. It uses a second-order model, factoring its parsing decisions over individual dependency arcs and parent-sibling relationships (two words which both have a dependency arc to a common parent). If S is a set of arcs used in the factorisation of the tree, the score of a S is determined by the dot product of its high-dimensional feature representation, and a weight vector as follows:

$$score(S) = \mathbf{w} \cdot \mathbf{f}(S)$$

$\mathbf{f}(S)$ encodes properties of the endpoints of the arcs, such as the words, POS tags, suffixes, or information incorporated from external resources, with most features being binary indicators that can be active (1) or inactive (0). The weight vector is determined

during training using a modified discriminative perceptron procedure, the Margin-Infused Relaxed Algorithm (MIRA).

MSTParser has been widely-used in the field, and we use it for its popularity and ease of modification; However, more recent work on graph-based parsing has yielded better performance though more complex algorithms such as variational inference (Martins et al., 2010), matrix tensor-factorisation (Lei et al., 2014), and cube pruning (Zhang and McDonald, 2014).

MSTParser draws its name from the notion of dependency parsing as searching for maximum-weight spanning trees in a weighted, directed graph. This generalises the projective and non-projective parsing algorithms commonly used by graph-based systems. Following previous work by Bansal and Klein (2011) and Pitler (2012a), we use the projective parsing mode of MSTParser for all of our experiments. We did not use the non-projective Chu-Liu-Edmonds algorithm described in McDonald et al. (2005b); for English, which generates primarily projective dependency trees, non-projective decoding algorithms perform significantly worse than projective ones. MSTParser uses Eisner's algorithm for projective decoding, which we will now describe.

3.2.1.1 Eisner's Algorithm for Projective Decoding

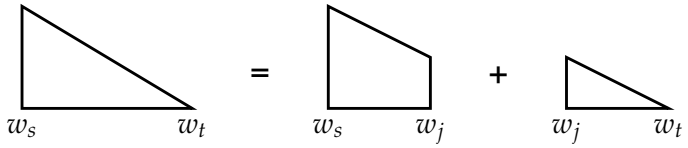
Projective graph-based parsing is strongly equivalent to context-free parsing. Given an extrinsic root node added to the far left of the sentence, there is an equivalence between the set of all possible projective dependency trees rooted at the new node and the set of all possible nested constituents (Kübler et al., 2009). Early graph-based methods used variants of the dynamic programming CKY algorithm for context-free grammar parsing (described in detail for CCG parsing in Chapter 5). The heart of the procedure is a *table* data structure C, where the item $C[s][t][i]$ records the best-scoring projective tree spanning the words $w_s \ldots w_t$ and rooted at w_i, where $s \leq i \leq t$. The item at $C[0][n][0]$, where n is the length of the sentence, represents the root of the highest-scoring tree over the entire sentence.

CKY-based dependency parsing fills the table based on the observation that any projective subtree rooted at w_i and spanning w_s to w_t must be composed of smaller subtrees. These smaller subtrees may be combined to build larger subtrees by progressively adding dependency arcs between them; that is, a tree spanning $w_s \dots w_{q-1}$ headed by $w_i, s \leq i < q$, and a tree spanning $w_q \dots w_t$ headed by $w_j, q \leq j \leq t$ can be combined into one tree spanning $w_s \dots w_t$ and headed by w_i by adding the dependency arc between w_s and w_j. Here, q is denoted as a *split point* between s and t. This gives rise to the following recurrence over the split points q and subtree heads j to calculate the best score for a projective tree headed by w_i between w_s and w_t:

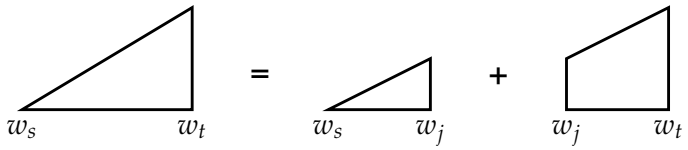
$$C[s][t][i] = \max_{s \leq q < t,\ s \leq j < t} \begin{cases} C[s][q][i] + C[q+1][t][j] + score(w_i, w_j) & \text{if } j > i \\ C[s][q][j] + C[q+1][t][i] + score(w_i, w_j) & \text{if } j < i \end{cases} \tag{3.1}$$

The CKY algorithm for dependency parsing has a runtime of $O(n^5)$, as it must fill $O(n^3)$ entries in the table, and for each one consider $O(n^2)$ split points and subtree heads. Eisner (1996b) proposed a substantially faster $O(n^3)$ algorithm which is equivalent to the CKY technique. Widely known as Eisner's algorithm, it is used in many parsers today, including MSTParser for projective dependency parsing. The algorithm is based on the observation that for a head w_s, its left dependents may be gathered independently from its right dependents without affecting the globally optimal score under the arc factorisation. This creates two types of subtrees (or *spans* in Eisner's notation):

- *complete* spans over $w_s \dots w_t$, which represent:
  - a span headed by w_s with a complete set of dependents on its right; or
  - a span headed by w_t with a complete set of dependents on its left;
- *incomplete* spans over $w_s \dots w_j$, which represent:
  - a span headed by w_s, with a dependency between w_s and w_j and potentially more dependents to be picked up to the right;

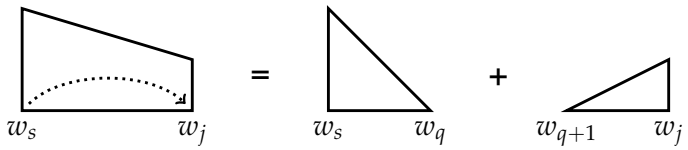


(a) Complete span headed by w_s

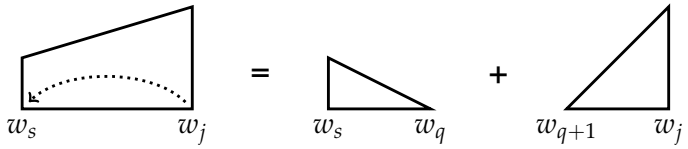


(b) Complete span headed by w_t

Figure 3.1: Left- and right headed complete spans in Eisner's algorithm.



(a) Incomplete span headed by w_s



(b) Incomplete span headed by w_j

Figure 3.2: Left- and right headed incomplete spans in Eisner's algorithm. The dashed arc is the dependency created by the operation.

– a span headed by w_j, with a dependency between w_j and w_s and potentially more dependents to be picked up to the left.

The root of the sentence is the head of a complete span with all of its dependents to the right. Each smaller span is recursively constructed from subsequently smaller spans; complete and incomplete spans are recursively constructed from one another.

Figures 3.1 and 3.2 depict how complete spans (represented by triangles) and incomplete spans (represented by trapezoids) are combined together to form larger spans. The operations creating incomplete spans from two complete spans form dependency

Algorithm 3.1 ***Eisner's Algorithm***

Require: a sentence $S = w_1, w_2, \ldots, w_n$
Require: a scoring function $\mathrm{S}(w_i, w_j)$ over the arc (w_i, w_j)
Require: a function $\textsc{label}(w_i, w_j)$ returning the best label over the arc (w_i, w_j)
Ensure: the best score for a tree over S in $E[0][n][L][C]$

Initialise all entries of $E[n+1][n+1][2][2] = 0.0$
Initialise all entries of $L[n+1][n+1][2] = null$
for all m from 0 to n **do**
    for all s from 0 to n **do**
        $t = s + m$
        if $t > n$ **then**
            break
        ▷ Create incomplete spans
        $L[s][t][R] \leftarrow \textsc{label}(w_t, w_s)$
        $E[s][t][R][I] \leftarrow \max_{s \leq q < t}(E[s][q][L][C] + E[q+1][t][R][C] + \mathrm{S}(w_t, w_s))$
        $L[s][t][L] \leftarrow \textsc{label}(w_s, w_t)$
        $E[s][t][L][I] \leftarrow \max_{s \leq q < t}(E[s][q][L][C] + E[q+1][t][R][C] + \mathrm{S}(w_s, w_t))$
        ▷ Create complete spans
        $E[s][t][R][C] \leftarrow \max_{s \leq j < t}(E[s][j][R][C] + E[j][t][R][I])$
        $E[s][t][L][C] \leftarrow \max_{s < j \leq t}(E[s][j][L][I] + E[j][t][L][C])$

arcs between the heads at opposite ends of the spans; forming a complete span does not yield any further dependency arcs.

The key optimisation in Eisner's algorithm is that the head of each subtree must be one of its peripheries. This allows the CKY factorisation over split points and intermediate heads to be collapsed into a factorisation over the type of span, and whether it is headed to the left or right. This relies on a table E, where $E[s][t][d][c]$ records the best-scoring projective span spanning $w_s \ldots w_t$, headed by w_s if $d = L$, or w_t if $d = R$. $c = I$ indicates that the span is incomplete, and $c = C$ indicates that the span is complete. Thus, $E[0][n][L][C]$ contains the score of the best parse over the entire sentence.

CKY and Eisner's algorithm may be easily extended to labeled dependency parsing by adding a secondary table L to record the best-scoring label for each arc in E. Algorithm 3.1 gives the pseudocode for the labeled Eisner's algorithm, describing how the

span combination operations in Figures 3.1 and 3.2 are used to find the best score of a tree spanning the input sentence.

McDonald and Pereira (2006) describes a straightforward second-order extension to Eisner's algorithm which is implemented in MSTParser, allowing it to include the most recent sibling attached in the score calculation at each step of the algorithm. Further work has developed more expressive extensions to Eisner's algorithm at the cost of increased computational complexity. Notably, Carreras (2007) describes an extended second-order model incorporating dependents of dependents, while Koo and Collins (2010) implements a third-order model considering heads of heads in addition to dependents.

3.2.2 ZPar: Transition-based Dependency Parsing

Transition-based techniques cast the dependency parsing problem as a series of decisions made over each word in a sentence, passing from left to right. The name stems from the *transition system* or *state machine* at the heart of such parsers, which encompasses a set of *states*, and transitions between them. Unlike graph-based parsers, which score dependency arcs directly, transition-based parsers are optimised over the series of transitions leading to the formation of a parse.

The most popular approach for dependency parsing uses states to encompass incomplete analyses of a sentence. A state $C = \langle T, B, A, s \rangle$, where T is a *stack* holding partially processed words, which may be used as the target or source of dependency arcs; B is a *buffer* holding the remaining words of the sentence which have not yet been processed in the state; A is the set of dependency arcs which have been created in the state; and s is the score of the state.

Each transition decision operates over two items in a state: the word at the top of the stack, and the word at the front of the buffer. The transition decides whether to move these items between the data structures, create arcs between them, or retire them entirely in the parse. Transition-based parsers are distinguished by the set of

transitions that they use, and the methodology they use to select the best transition at each stage of the parsing process.

ZPar (Zhang and Clark, 2011b) is a labelled transition parser with four transitions:

- ***Shift*** the front word of the buffer to the top of the stack;
- ***Reduce*** the top word of the stack, removing and discarding it;
- ***ArcRight*** to create a dependency arc headed by the top of the stack to the front word of the buffer, then ***Shift***;
- ***ArcLeft*** to create a dependency arc headed by the front word of the buffer to the top of the stack, then ***Reduce***;

This system of transitions is known as *arc-eager* projective shift-reduce parsing, and it was popularised by Nivre and Scholz (2004).

In ZPar, labels are assigned by parameterising each ***ArcRight*** and ***ArcLeft*** transition with a dependency label, meaning that there are $2k + 2$ transitions in total, where k is the number of dependency labels. As a sentence is parsed, each word must be shifted onto the stack once and reduced from the stack once, meaning that the transition model runs in linear time. This is asymptotically faster than the cubic-time Eisner's algorithm, contributing to the growth in popularity of transition-based techniques.

ZPar chooses transitions using an averaged perceptron model (Collins, 2002). At each decision point, the parser extracts features based on the current parser state, primarily from the words on top of the stack and at the front of the buffer. These features are used to score all possible transitions from the current state, with the most likely transition under the model selected and applied. Early transition-based parsers used one-best transition selections (Yamada and Matsumoto, 2003; Nivre and Scholz, 2004), but this has the disadvantage of not allowing the parser to recover from an incorrect transition which may score highly based on its local context, but is less optimal later in the parsing process.

By contrast, ZPar uses *beam search*, and keeps the top M scoring transitions and states in a beam. At each decision point, ZPar calculates the best transition that can

Algorithm 3.2 ***Transition-based Beam Search Parsing***

Require: a sentence $S = w_1, w_2, \ldots, w_n$
Require: a set of transitions $\mathrm{T} = \tau_1, \tau_2, \ldots$, each of which takes a state and returns a new state
Require: a beam size M
Ensure: the best final state for a dependency tree over S

$T \leftarrow$ empty stack
$B \leftarrow$ buffer filled with words of S in order
$A \leftarrow$ empty set of arcs
beam $\leftarrow$ priority queue containing C, sorted descending by score
agenda $\leftarrow$ empty priority queue, sorted descending by score
$\triangleright$ Create the initial state
$C \leftarrow (T, I, A, 0)$
for all states C in *beam* **do**
    for all transitions $\tau \in \mathrm{T}$ **do**
        $C' \leftarrow \tau(C)$
        if C' is not *null* **then**
            add C' to *agenda*
    $(T', B', A', s') \leftarrow$ the top item in *agenda*
    if there is one item in T' and no items in B' **then**
        return the top item in *agenda*
    beam $\leftarrow$ the top M items in *agenda*

be applied to each state, retaining the top M overall for the next step. Beam search is an approximation that has the advantage of retaining some degree of backtracking capability without causing a combinatorial explosion in the number of states which must be tracked. Zhang and Clark (2011b) find that a beam of size 64 offers a good combination between parsing speed and accuracy for English dependency parsing.

Algorithm 3.2 gives the pseudocode for transition-based beam search parsing.

3.3 Analysing Parser Errors

Dependency representations have long been associated with more analytical parser evaluation metrics. Lin (1995) and Briscoe et al. (2002) argue that constituency parser evaluation is overly focused on irrelevant details of tree topology, and advocate for

parser evaluation based on a conversion to dependencies. Rimell et al. (2009) proposed a parser evaluation based on unbounded dependencies, which may have any number of intervening words between the head and the dependent. However, these evaluations do not explain why particular attachments are incorrect, or whether the presence of one attachment results in other errors in the parse.

Nilsson (2009) categorises dependency parser errors using two metrics: error propagation, and error clustering. Both measures attempt to capture whether an initial error causes further errors later in a sentence. Error propagation divides the sentence into two: a *pre* section consisting of all tokens up to and including the first error, and a *post* section with all of the remaining tokens. The metric then compares the proportion of errors in the pre and post sections. Error clustering measures the average error rate for tokens immediately following an error. While informative, these measures only imply a relationship between errors – a relationship that only exists in the forward direction. This linearity is not guaranteed to hold in more complex parsing models, where features can be extracted over higher-order relationships.

McDonald and Nivre (2011) perform an in-depth comparison of the graph-based MSTParser and transition-based MaltParser. However, MaltParser uses support vector machines to deterministically predict the next transition, rather than storing the most probable options in a beam like ZPar. Additionally, they do not focus on the cascading impact of errors, and instead concentrate on higher-level error classification (e.g. by POS tag, labels and dependency lengths) in lieu of examining how the parsers respond to forced corrections.

Nivre et al. (2014) describe several uses for arc-level constraints in transition-based parsing. However, these applications focus on improving parsing accuracy when constraints can be readily identified, e.g. imperatives at the beginning of a sentence are likely to be the root. We focus our constraints on evaluation, attempting to identify important sources of error in dependency parsers.

Our constraint-based approach shares similarities to oracle training and decoding methods, where an external source of truth is used to verify parser decisions. An oracle source of parser actions is a necessary component for training transition-based parsers (Nivre, 2009). Oracle decoding, where a system is forced to produce correct output if possible, can be used to assess its upper performance bounds (Ng and Curran, 2012).

Constraining the parser's internal search space is akin to an optimal pruning operation — a common technique for increasing parsing speed. Charniak and Johnson (2005) use a coarse-to-fine, iterative pruning approach for efficiently generating high-quality n-best parses for a discriminative reranker. Koo and Collins (2010) and Rush and Petrov (2012) use a similar coarse-to-fine pruning algorithm with dependency parsing; the former prunes arcs to decrease training times, while the latter prunes vine grammars (Eisner and Smith, 2005) to accelerate graph-based dependency parsing, achieving parsing speeds close to linear-time transition parsers despite encoding more complex features.

Pruning can also be implemented using single-pass, rather than iterative approaches. These techniques typically rely on fast tagging or classification stages to reduce the search space of the more expensive parsing step. Roark and Hollingshead (2008) classify word positions based on whether they can grammatically begin or end constituents. Based on this classifier, they close chart cells to new entries, reducing computational complexity and increasing speed in the Charniak parser. Roark and Hollingshead (2009) extend this work, applying hard constraints from an initial tagging pass over a sentence to the chart itself, further improving parsing speed. Supertagging (Clark and Curran, 2007a) and chart pruning (Zhang et al., 2010) have also been used to constrain the search space of a CCG parser, removing unlikely or forbidden spans from repeated consideration. In our work, we use pruning not for parsing speed, but evaluation, and so we prune items based on gold-standard constraints rather than heuristics.

Kummerfeld et al. (2012) perform a comprehensive classification of constituency bracket errors and their cascading impact, and their work is the most philosophically

similar to ours. They associate groups of bracket errors in the parse with abstract error types, and identify the tree operations that repair these error types. Specifically, the repair operations include the insertion, deletion, or substitution of nodes in the parse tree, and their error categories include:

PP attachment: any error where the repair operation moves a PP, or the incorrect bracket is over a PP;

NP attachment: an error where an NP node must be moved in the tree, particularly for appositional mistakes and incorrect attachments inside verb phrases;

modifier attachment: any error involving missing or incorrectly placed adjective and adverb nodes;

clause attachment: any error which requires the movement of an S node;

unary: any error involving a unary production;

coordination: any error where a coordinator is an immediate sibling of the nodes being moved, or is one of the outermost nodes being moved;

NP internal structure: errors involving node types existing only within NPs;

different label: an error where the node structure is correct, but its label is incorrect;

single word phrase: an error which does not fall into a different category, and spans a single word;

other: all other errors.

The error types in a particular parser's output are identified through a heuristic procedure that repeatedly applies the repair operation that removes the largest number of bracket errors. The list of error types associated with the operations form the canonical set of errors made in the parse. Kummerfeld et al. (2012) use this to perform an in-depth investigation of the errors made by a wide variety of constituency parsers.

The weakness of the Kummerfeld et al. (2012) approach is the assumption that, when an error is detected, repairing that error will have no impact on the remainder of the sentence. However, it is possible that the parser may not perform the repair in full, or be incapable of constructing the repaired tree due to limitations in its grammar or

Parser	UAS	LAS
MSTParser	91.3	87.5
ZPar	91.7	89.3

Table 3.1: Baseline unlabeled and labeled attachment scores on Stanford dependencies over OntoNotes WSJ section 22.

parsing model. Cascading changes also cannot be ruled out, depending on whether the repair affects head words or phrase boundaries. Parsing is a complex process, and it is difficult to assume that an ad hoc, though principled, rearrangement of nodes will always occur perfectly. The reverse is also true: repairing one type of error completely may cause the model to select a different attachment elsewhere, which can improve or worsen performance.

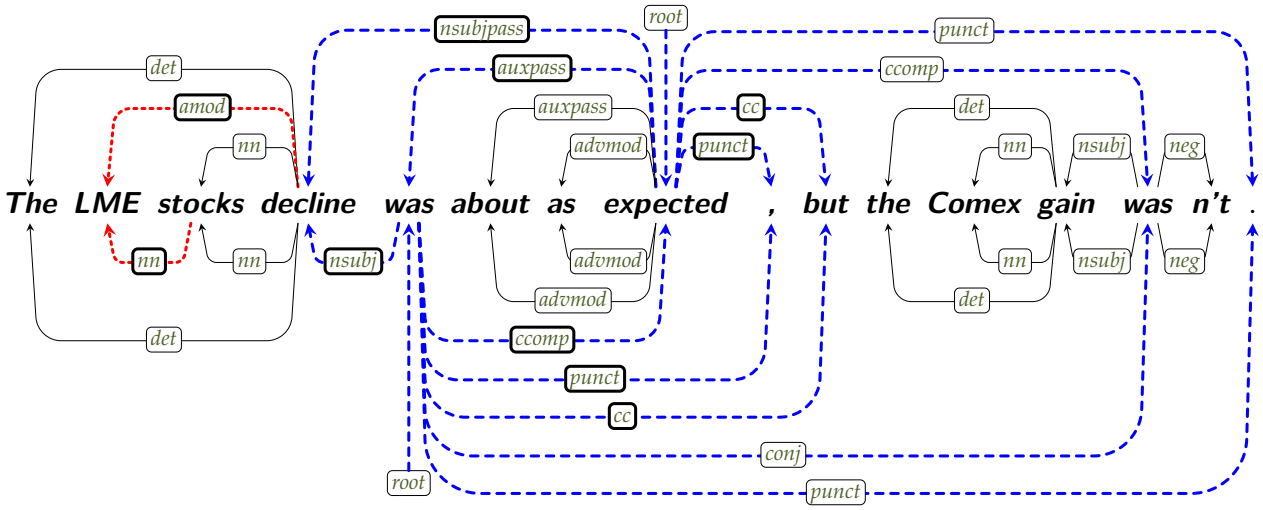
3.4 Motivation

Table 3.1 summarises the performance of MSTParser and ZPar on Stanford dependencies over OntoNotes 4 WSJ section 22. ZPar performs slightly better than MSTParser on unlabeled attachments, and substantially better in labeled attachments. However, these numbers do not show what types of errors are being made by each parser, what errors remain to be addressed, or hint at what underlying problems cause each error.

Parse 3.1 depicts a WSJ section 22 sentence as parsed by MSTParser, and the gold parse. The unlabeled attachment metric reports a score of 47.1%, with 8 of 17 arcs correct. By contrast, ZPar (parse not shown) scores 94.1%, with the sole attachment error being on *LME* (as with MSTParser).

While there are nine incorrect arcs overall, MSTParser seems to have made only two underlying errors:

- *LME* attached to *decline* rather than *stocks* (NP internal). Correcting this repairs one error;



Parse 3.1: MSTParser output (top) and the gold parse (bottom) for a WSJ 22 sentence. MSTParser has produced two independent errors: an NP bracketing error (red, dotted), and an incorrect root (blue, dashed).

- *expected* being chosen as the root rather than *was*. Correcting the root and moving all attachments to it from the old root repairs the remaining eight errors.

Intuitively, it seems that the impact of the NP error is limited — the incorrect structure is isolated within the NP and does not cause any other errors in the sentence. By contrast, the root selection error has a substantial impact on the second half of the sentence, causing a misplaced subject, misattached punctuation, and incorrect coordination. It appears that these cascaded errors have occurred *because of* the incorrect root, rather than in isolation from it.

What we do not know is whether these intuitions actually hold in the context of the parsing process. Many dependency parsers, including MSTParser and ZPar, construct trees by repeatedly combining smaller tree fragments together, beginning from individual words until a spanning analysis is found. The local context used by Eisner's algorithm or the arc-eager transition process means that only a small window of information is used to make each arc decision. An error in one part of the tree may have no influence on a different part of the tree. Alternatively, if the feature representation and decoding algorithm is sufficiently rich, errors may exert long-range

influence — particularly if there are higher-order features or algorithmic constraints such as projectivity over the tree. Given the complex interactions in parsing algorithms, we wish to define a method to repair various error types in isolation without otherwise assuming the subsequent actions of the parser.

3.5 Constraint-driven Analysis of Parser Performance

We aim to investigate the way each parser reacts when certain errors in the tree are corrected. To do this, we force each parser to select the correct head and label for certain words, but otherwise allow it to construct its best parse. Given a set of *constraints*, each of which lists a word with the head and label to which it must be attached, we can investigate two measures:

1) the number of errors *directly corrected* by the set of constraints. We call this the *constrained accuracy impact*;

2) the *indirect impact of the constraints*, including the number of errors *indirectly corrected*, and the number of correct arcs *indirectly destroyed*. We call this the *cascaded accuracy impact*.

The constrained accuracy impact tells us how often the parser makes errors in the set of words covered by the constraints, measured using the per-word attachment accuracy calculated over only the words covered by the constraint set.

The cascaded accuracy impact is the less predictable figure, as it tells us what effect the errors made over the constrained set of arcs have over the rest of the sentence. It is the *influence* of the set of constraints over the other attachments, and answers the question of what the parser would do when it is forced to correctly attach constrained fillers. This influence may be felt through projectivity requirements, or changes in the context used for other parsing decisions.

The core of our procedure is adapting each parser to accept constraints specifying the correct head and label for certain words in a sentence. Unlike Kummerfeld et al.

(2012), whose post-processing error identification assumes the parser would respond perfectly to each repair, our constraint methodology makes no assumptions about what the parser will do when an error is repaired.

3.5.1 Experimental Setup

We use the OntoNotes 4 release of the PTB WSJ data and the English Web Treebank, as described in Section 2.4.1. This data is consistent with the SANCL 2012 Shared Task (Petrov and McDonald, 2012), and means that our procedure can be easily ported to the many systems participating in the task.

We changed all marker tokens in the corpora (e.g. -LRB- and -LCB-) to their equivalent unescaped punctuation marks ([and {) to ensure correct evaluation. The organisers of the SANCL task provided both corpora converted to basic Stanford dependencies using the Stanford Parser v2.0.[1]

The WSJ sections 02-21 are used for training each parser, while section 22 is used as the basis of our evaluation procedure. The gold-standard annotations of section 22 are the source of the constraints and the data used to evaluate each parser. We do not use any of the Web Treebank data to train the parsers, but we concatenate the development splits of the five Web Treebank domains as supplied by the SANCL 2012 Task to implement our evaluation on web text.

We generated POS tags for the corpus using MXPOST, a maximum-entropy Markov tagger (Ratnaparkhi, 1996). We trained MXPOST over the gold-standard POS tags from OntoNotes 4 WSJ sections 02-21 on default settings, and used the resulting model to tag the development data for both corpora. We used ten-fold jackknife training as follows to tag the training data:

- MXPOST was trained using all of the gold-standard OntoNotes 4 WSJ training data, except the first two sections;

[1] http://nlp.stanford.edu/software/lex-parser.shtml

- the resulting model was used to tag the first two sections of the training data;
- this process was repeated in turn for the next two sections of the training data, totalling ten splits and training sections overall.

MSTParser uses features of both coarse-grained POS tags and fine-grained POS tags, both of which were provided by the CoNLL-X Shared Task on Multilingual Dependency Parsing (Buchholz and Marsi, 2006). We approximate the coarse-grained POS tags by taking the first character of the POS tag assigned to the word by MXPOST, a technique also used by Bansal et al. (2014).[2]

3.5.2 Error Classes

Following Kummerfeld et al. (2012), we define meaningful error classes grouped with the operations that repair them. In the dependency parser setting, error classes are groups of Stanford dependency labels, rather than groups of node repair operations. The Stanford labels provide a rich distinction in NP internal structure, clauses, and modifiers, and map very well to the error categories of Kummerfeld et al. (2012), allowing us to avoid excessive heuristics in the mapping process. Our technique can be applied to other dependency schemes such as LTH (Johansson and Nugues, 2007) by defining new mappings from labels to error types.

The difficulty of the mapping task depends on the intricacies of each formalism. For example, the major challenge with LTH dependencies is the enormous skew towards the nominal modifier *NMOD* label. This label occurs 11,335 times in the WSJ section 22, more than twice as frequently as the next most frequent punctuation *P*. By contrast, the most common Stanford label is punctuation, at 4,731 occurrences. The *NMOD* label is split into many smaller, but more informative nominal labels in the Stanford scheme, making it better suited for our goal of error analysis.

[2] Mohit Bansal, p.c.

The label grouping was performed with reference to the Stanford dependencies manual v2.04 (de Marneffe and Manning, 2008a, updated 2012). For each error class, we generate a set of constraints over section 22 for all words with a gold-standard label in the category associated with the class. Our types are defined as follows:

NP attachment: any label specifically attaching an NP, includes *appos*, *dobj*, *iobj*, *nsubj*, *nsubjpass*, *pobj*, and *xsubj*.

NP internal: any label marking nominal structure (not including adjectival modifiers), includes *abbrev*, *det*, *nn*, *number*, *poss*, *possessive*, and *predet*.

PP attachment: any label attaching a prepositional phrase, includes *prep*. Also includes *pcomp* if the POS of the word is TO or IN.

Clause attachment: any label attaching a clause, includes *advcl*, *ccomp*, *csubj*, *csubjpass*, *purpcl*, *rcmod*, and *xcomp*. Also includes *pcomp* if the POS of the word is not TO or IN.

Modifier attachment: any label attaching an adverbial or adjectival modifier, includes *advmod*, *amod*, *infmod*, *npadvmod*, *num*, *partmod*, *quantmod*, and *tmod*.

Coordination attachment: *conj*, *cc*, and *preconj*.

Root attachment: the *root* label.

Punctuation attachment: the *punct* label.

Other attachment: all other Stanford labels, specifically *acomp*, *attr*, *aux*, *auxpass*, *complm*, *cop*, *dep*, *expl*, *mark*, *mwe*, *neg*, *parataxis*, *prt*, *ref*, and *rel*.

For example, Root constraints specify which words in each sentence are roots, while PP constraints specify which words are heads of prepositions.

One deficiency of our implementation is that we apply constraints to all arcs of a particular error type in each sentence, and do not attempt to isolate multiple instances of the same error class in a sentence. We do this since applying a single constraint to a sentence one at a time would require modifications to the standard evaluation regime.

Table 3.2 gives the distribution of constraints over each error class in WSJ section 22 and the combined development sets of the English Web Treebank. In the WSJ, Root

	WSJ 22		EWT Dev	
Constraint Type	Freq.	%	Freq.	%
NP attachment	6789	21.2	27864	21.7
NP internal	6682	20.8	19171	14.9
Modifier attachment	4274	13.3	14776	11.5
Punctuation attachment	3724	11.6	14413	11.2
Other attachment	3323	10.4	18021	14.1
PP attachment	2974	9.3	10885	8.5
Coordination attachment	1628	5.1	8293	6.5
Clause attachment	1362	4.2	6501	5.1
Root attachment	1336	4.2	8314	6.5
Total	32092	100.0	128238	100.0

Table 3.2: The number of constraints (dependencies) in each error type over the OntoNotes 4 WSJ section 22 and the combined development sections of the Web Treebank. The total number of sentences in each is the number of Root attachment constraints.

attachment and Clause attachment are the smallest classes, with 1,336 constraints (one for each sentence in the data) and 1,362 constraints each. NP attachment and NP internal are the two largest classes, with 6,789 and 6,682 constraints respectively. The remaining constraints are spread roughly evenly between the other classes.

For the Web Treebank, the conversion to Stanford dependencies has resulted in more constraints in the Other attachment class. Table 3.3 gives the breakdown of label frequency in the class across the WSJ and Web Treebank, showing that there is a marked increase in *acomp* (adjectival complements), *attr* (attributives), *aux* (auxiliary verbs) and *neg* (negation) labels between the newswire and web domains. This reflects the typical style of web text, particularly the more informal genres such as question-answers, reviews, and newsgroups.

There are substantially fewer NP internal constraints in the Web Treebank compared to the WSJ data, suggesting a reduced complexity in NP structure. There is also a higher proportion of Root attachment constraints, signifying a shorter average sentence length. Coordination attachment and Clause attachment constraints have also increased as a proportion of the overall constraint set, while all other error classes have declined

	WSJ 22		EWT Dev	
Label	Freq.	%	Freq.	%
dep	1031	31.0	5549	30.8
aux	1007	30.3	5671	31.5
auxpass	231	7.0	851	4.7
mark	212	6.4	1017	5.6
acomp	184	5.5	1500	8.3
neg	153	4.6	1037	5.8
attr	151	4.5	953	5.3
complm	128	3.9	561	3.1
prt	104	3.1	385	2.1
parataxis	43	1.3	53	0.3
mwe	37	1.1	113	0.6
expl	29	0.9	249	1.4
rel	12	0.4	56	0.3
cop	1	0.0	26	0.1
Total	3323	100.0	18021	100.0

Table 3.3: The number of constraints per label of the Other attachment error class over OntoNotes 4 WSJ section 22 and the combined development sections of the Web Treebank.

slightly. However the overall similarity in distribution to newswire text reinforce that English has distributional regularities in syntax even across these diverse domains.

3.6 Applying Constraints in Parsers

In this section, we describe how we implement constraints in MSTParser and ZPar.

3.6.1 MSTParser implementation

We enforce constraints using MSTParser's projective decoding mode by modifying when incomplete spans (which create dependencies) are permitted to form in Eisner's algorithm. Incomplete spans are only allowed from constrained words to their correct heads with the correct labels. Any incomplete span between an incorrect head and the constrained words is forbidden. The algorithm is forced to choose the constrained spans as it builds the parse, taking advantage of the single head assigned per word. Thus, as long as the constraints are consistent, they have no impact on the parser's coverage as all other possible head selections are considered.

This approach is similar to that employed by Rush and Petrov (2012), who use a coarse-to-fine approach to filter the words available for combination in Eisner's algorithm. However, while they employ a linear-time vine parser to eliminate unlikely words from consideration in the creation of incomplete spans, we explicitly forbid any spans which are inconsistent with the set of constraints. Algorithms 3.3 and 3.4 describe the constrained decoding algorithm.

All MSTParser experiments in this thesis are run with the following arguments: *order:2*, *training-k:5*, *iters:10*, and *loss-type:nopunc*, mirroring the configuration used by Bansal and Klein (2011).

Algorithm 3.3 ***constrained-label***

Require: a list of arc constraints $constraints = [(w_h, w_f, l'), \dots]$
Require: the dependency label l to be assigned to the arc (h, f)
Require: a head index h and filler index f
Ensure: *null* if (h, f) violates a constraint, l if there is no constraint on (h, f), or the constrained label l' otherwise

```
1 if constraints contains a constraint with f as the filler then
2     (head, filler, l') ← the constraint where f is the filler
3     if head == h then
4         return l'
5     else
6         return null
7 else
8     return l
```

Algorithm 3.4 ***Constrained Eisner's Algorithm***

Require: a sentence $S = w_1, w_2, \dots, w_n$
Require: a scoring function $\mathsf{S}(w_i, w_j)$ over the arc (w_i, w_j)
Require: a list of arc constraints $constraints = [(w_h, w_f, label), \dots]$
Ensure: the best score for a tree over S in $E[0][n][L][C]$ containing all arcs in l

```
1  Initialise all entries of E[n+1][n+1][2][2] = 0.0
2  Initialise all entries of L[n+1][n+1][2] = null
3  for all m from 0 to n do
4      for all s from 0 to n do
5          t = s + m
6          if t > n then
7              break
8          ▷ Create incomplete spans if they are permissible
9          label ← constrained-label (constraints, LABEL(w_t, w_s), t, s)
10         if label is not null then
11             L[s][t][R] ← label
12             E[s][t][R][I] = max_{s≤q<t}(E[s][q][L][C] + E[q+1][t][R][C] + S(w_t, w_s))
13         label ← constrained-label (constraints, LABEL(w_s, w_t), s, t)
14         if label is not null then
15             L[s][t][L] ← label
16             E[s][t][L][I] = max_{s≤q<t}(E[s][q][L][C] + E[q+1][t][R][C] + S(w_s, w_t))
17         ▷ Create complete spans
18         E[s][t][R][C] = max_{s≤j<t}(E[s][j][R][C] + E[j][t][R][I])
19         E[s][t][L][C] = max_{s<j≤t}(E[s][j][L][I] + E[j][t][L][C])
```

3.6.2 ZPar implementation

Algorithm 3.5 ***ConstrainedArcLeft***

Require: a dependency label l to be assigned to the arc
Require: a list of arc constraints $constraints = [(w_h, w_f, label), \dots]$
Require: a parser configuration $C = (T, B, A, s)$
Ensure: a parser configuration $C' = (T', B', A', s')$ if the new arc is permitted and compatible with l, otherwise *null*

```
1  if there are no more words in T or B then
2      return null
3  h ← the index of the word at the front of the buffer B
4  f ← the index of the word on top of the stack T
5  label ← constrained-label (constraints, l, h, f)
6  if label is not null then
7      (T', B', A') ← CLONE(T, B, A)
8      add (h, f, label) to A'
9      pop h from T'
10     s' ← s + SCORE(T', B', A')
11     return C' = (T', B', A', s')
12 else
13     return null
```

We enforce constraints for ZPar in a similar way to that of Nivre et al. (2014), who apply span-level and arc-level constraints to an arc-eager dependency parser. We disallow any ***ArcLeft*** or ***ArcRight*** action creating an arc that conflicts with a provided constraint. However, unlike MSTParser, these constraints may have an impact on the parser's coverage due to their interaction with beam search. Even when a constraint is fulfilled by an arc action, it may not be scored highly enough to stay within the beam. It is also not guaranteed that each action will consume a word in the sentence. This means that the beam may be filled with states that *could eventually* satisfy a constraint, and so cannot be eliminated, even if the parser will never score the eventual states with constraints high enough for them to stay in the beam. Thus, it is possible for the parser to evict all states containing an enforced arc from the beam, leaving only states which

Algorithm 3.6 ***ConstrainedArcRight***

Require: a dependency label l to be assigned to the arc
Require: a list of arc constraints $constraints = [(w_h, w_f, label), \dots]$
Require: a parser configuration $C = (T, B, A, s)$
Ensure: a parser configuration $C' = (T', B', A', s')$ if the new arc is permitted and compatible with l, otherwise *null*

```
1  if there are no more words in T or B then
2      return null
3  h ← the index of the word on top of the stack T
4  f ← the index of the word at the front of the buffer B
5  label ← constrained-label (constraints, l, h, f)
6  if label is not null then
7      (T', B', A') ← CLONE((T, B, A))
8      add (h, f, label) to A'
9      push h onto T' and remove it from B'
10     s' ← s + SCORE(T', B', A')
11     return C' = (T', B', A', s')
12 else
13     return null
```

will not satisfy the constraints. When this happens, no correct analysis is retrievable, and the parser will fail to find an analysis for the sentence.

ZPar also implements manually-crafted rules on which POS tags are allowed to attach to one another, in order to reduce the search space and improve accuracy. In the case where we specify a constraint enforcing an arc that is not permitted by the parser's POS tag constraints, the correct arc will be disallowed, and the parser will fail to parse the sentence.

Counteracting these problems is not possible without compromising our analysis, or altering the ordinary behaviour of the parser. We can choose to make the parser fall back on the baseline parse when it fails to find an analysis using constraints. However, this affects the comparison between MSTParser and ZPar, as the former will seem to accept more constraints due to its higher coverage. Alternatively, the POS tag rules may be removed, and the beam size increased from its default value of 64 to reduce the

number of "dead ends" in the beam. This second option has the downside of changing the decisions of the parser, which is undesirable for our analysis.

All ZPar experiments in this thesis are trained with 10 iterations of the averaged perceptron algorithm and a beam size of 64.

We ensured that our modifications were working correctly for both parsers through two experiments. First, we utilised zero constraints, and verified that the output for each parser was identical to the baseline. We then utilised every possible constraint, and verified that the output was 100% correct.

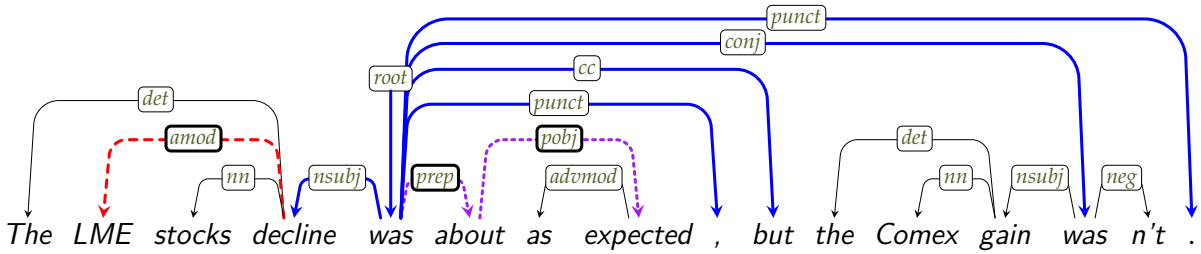
3.7 Evaluation

We implement a custom evaluation script to facilitate a straightforward comparative analysis between the cascaded and constrained output. The script is based on and produces identical scores to `eval06.pl`, the official evaluation for the CoNLL-X Shared Task on Multilingual Dependency Parsing (Buchholz and Marsi, 2006). We ignore punctuation as defined by `eval06.pl` in our evaluation; experiments with constraints over punctuation tokens constrain those tokens in the parse, but ignore the punctuation dependencies during evaluation.

We run the modified parsers over WSJ section 22 with and without each set of constraints. We examine the overall unlabeled and labeled attachment scores (UAS and LAS), as well as identifying the contribution to the overall UAS improvement from constrained accuracy impact and cascaded accuracy impact. When examining MSTParser and ZPar side by side, we evaluate only over the sentences covered by ZPar to ensure a fair comparison.

3.7.1 Constrained and Cascaded Arcs

When comparing the baseline parse of a sentence to one where constraints have been applied, we can identify three disjoint groups of dependencies:



Parse 3.2: MSTParser output for the sentence in Parse 3.1, where the *root* arc is forced to its correct position. The NP error is unaffected by the constraint (dashed, red), six attachment errors are repaired (solid, blue), and two new errors are introduced (dotted, purple).

- *unchanged dependencies*, which are identical in the baseline parse and the constrained parse. These dependencies were either already correct in the baseline (and hence applying a constraint to them had no effect), or were not constrained;
- *constrained dependencies*, which were incorrect in the baseline, but have been forced to be correct in the constrained parse;
- *cascaded dependencies*, which have changed compared to the baseline. There are two cases:
  - the dependency was incorrect in the baseline, but is corrected in the constrained parse;
  - the dependency was correct in the baseline, but has become incorrect in the constrained parse.

Parse 3.2 and Table 3.4 show the impact of applying constraints on tokens with various labels to MSTParser for the sentence in Parse 3.1. We can see that enforcing the gold *nn* arc between *decline* and *LME* repairs the NP error, but does not affect any of the other errors. This matches our intuition, and the constraint has resulted in one constrained dependency, and sixteen unchanged dependencies.

Conversely, enforcing the gold *root* arc does not affect the NP error, but repairs nearly every other error in the parse. Unfortunately, the constrained *root* arc introduces

Constraints	✓	✗	Remaining Errors
None	8	9	see Parse 3.1
nn	9	8	All except *decline* → *LME*
root	14	3	*decline* → *LME* *about* → *expected* *was* → *about*
punct	14	3	*decline* → *LME* *about* → *expected* *was* → *about*
ccomp	16	1	*decline* → *LME*

Table 3.4: Correct and incorrect arcs, and the remaining errors after applying various sets of constraints to the sentence in Parse 3.1.

two new errors, with the parser incorrectly attaching the clausal complement headed by *expected* and the modifier headed by *about*. There is one constrained dependency, eight unchanged dependencies, and eight cascaded dependencies, with six of the cascaded dependencies being correct, and two being incorrect. While our intuition regarding the underlying root attachment error repair seems to have held under the context of the parsing process, the parser has also introduced new errors.

Enforcing the correct attachment for the two punctuation marks has an identical impact to enforcing the *root* dependency. As punctuation often attaches to phrasal heads or roots in dependency formalisms, these constraints act as a proxy for the root constraint in this sentence.

Enforcing the correct *ccomp* arc in isolation rather than the *root* arc leads to MSTParser producing the full correct analysis for the second half of the sentence (though again, this single constraint does not repair the separate NP error). There is one constrained dependency, eight cascaded correct dependencies, and eight unchanged dependencies due to the constraints. This runs counter to our initial expectations; spec-

ifying the *ccomp* arc seems to have indirectly specified the root word in MSTParser as well as avoided the additional errors that constraining the root word alone introduced.

This example demonstrates why we have chosen to implement our evaluation as a set of constraints in the parser, rather than Kummerfeld et al. (2012)'s post-processing approach. We cannot know how the parser will react when repairing errors, and the impact of the parser model is difficult to quantify after parsing is completed. While it seemed that the incorrect root attachment was the underlying cause of most of the errors in the parser, applying a constraint to the clausal complement was actually what corrected the errors entirely.

3.8 Comparing MSTParser and ZPar

3.8.1 Newswire

Tables 3.5 and 3.6 summarise our results on MSTParser and ZPar over WSJ section 22. Each set of results is calculated over the sentences covered by ZPar to ensure that they are comparable (MSTParser has perfect coverage due to its graph-based nature). We primarily discuss UAS results in this thesis, as LAS results are largely consistent.

The UAS of constrained arcs in each experiment is the expected 100%. Effective constraints is the number of constraints which repaired an error in the baseline, and the effective constraint percentage is the percentage of the constraints applied which repaired an error. These two metrics show how many mistakes the baseline parser made in the error class associated with the constraints; thus the percentage can also be viewed as the error rate. The error displacement is the average number of words that effective constraints moved an attachment point. A large error displacement indicates that incorrect arcs were moved a longer distance in the sentence when constraints were applied — the parser had misplaced the attachment by a longer distance.

The overall ΔUAS improvement is divided into Δc, the constrained impact, and Δu, the cascaded impact. The ratio of these two values with respect to each other shows

Error class	cover	eff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
Baseline	100.0	-	-	-	91.3	87.5	-	-	-
NP attachment	95.6	312	4.9	5.2	94.1	90.7	2.3	1.2	1.1
NP internal	98.2	206	3.2	2.8	92.6	89.2	1.1	0.8	0.3
Modifier attachment	96.8	321	7.9	3.8	93.4	90.3	1.7	1.2	0.5
PP attachment	98.3	378	13.1	4.3	93.2	89.5	1.7	1.4	0.3
Coordination attachment	97.7	238	16.0	5.8	92.9	89.5	1.3	0.9	0.4
Clause attachment	96.7	228	17.9	6.9	93.0	89.6	1.4	0.9	0.5
Root attachment	99.1	77	5.8	9.3	92.2	88.3	0.8	0.3	0.5
Punctuation attachment	93.2	469	14.2	7.4	93.9	89.9	1.8	0.1	1.7
Other attachment	94.3	210	7.0	6.1	93.5	90.8	1.4	0.8	0.6
All attachments	98.5	2912	9.3	5.8	100.0	100.0	8.6	8.6	0.0

Table 3.5: The coverage, effective constraints and percentage, error displacement, UAS, LAS, ΔUAS over the corrected arcs, and the constrained and cascaded Δ for MSTParser over the WSJ 22 sentences covered by ZPar.

Error class	cover	eff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
Baseline	100.0	-	-	-	91.7	89.2	-	-	-
NP attachment	95.6	277	4.3	4.8	94.9	92.7	2.4	1.0	1.4
NP internal	98.2	197	3.0	3.0	93.2	91.1	1.2	0.7	0.5
Modifier attachment	96.8	303	7.5	3.9	94.0	92.3	1.8	1.1	0.7
PP attachment	98.3	357	12.4	3.9	93.8	91.4	1.7	1.3	0.4
Coordination attachment	97.7	240	16.2	5.8	93.5	91.1	1.3	0.9	0.4
Clause attachment	96.7	166	13.0	5.6	93.4	91.2	1.2	0.6	0.6
Root attachment	99.1	57	4.3	9.9	92.4	89.9	0.5	0.2	0.3
Punctuation attachment	93.2	430	13.0	7.3	94.5	92.1	1.6	0.2	1.5
Other attachment	94.3	187	6.3	5.5	94.2	92.7	1.3	0.7	0.6
All attachments	98.5	2760	8.8	5.8	100.0	100.0	8.0	8.0	0.0

Table 3.6: The coverage, effective constraints and percentage, error displacement, UAS, LAS, ΔUAS over the baseline, and the constrained and cascaded Δ for ZPar over WSJ 22.

the influence of each error class; the larger Δu is relative to Δc, then the greater the cascading impact from applying constraints.

It is important to note that a parser may make a substantial number of mistakes on a particular error class (large effective constraint percentage), but correcting those mistakes may have very little cascading impact (small Δu), limiting the overall ΔUAS improvement. Conversely, there may be a class with a small effective constraint percentage, but a large ΔUAS due to a large cascading impact from the corrections, or simply because the class contains more constraints.

When applying all constraints, ZPar has a 8.8% effective constraint percentage compared to 9.3% for MSTParser. This is directly related to the UAS difference between the parsers. Aside from coordination, where the parsers made a nearly identical number of errors, ZPar is more accurate across the board. It makes substantially fewer mistakes on clause attachments, punctuation dependencies, and NP attachments, whilst maintaining a small advantage across all of the other categories.

The relative rank of the effective constraint percentage per error category is similar across the parsers, with PP attachment, punctuation, modifiers, and coordination recording the largest number of effective constraints, and thus the most errors. This illustrates that the behaviour of both parsers is very consistent, despite one considering every possible attachment point, and the other using a linear transition-based beam search. ZPar is able to make fewer mistakes across each error category, suggesting that the beam search pruning is maintaining more desired states than the graph-based parser is able to rank during its search.

ZPar's coverage is 98.5% when applying all constraints. However, as the number of constraints is reduced, coverage also drops. This seems counter-intuitive, but applying more constraints has the effect of eliminating more undesired states, leaving more space in the beam for satisfying states. Reducing the number of constraints permits more states which do not yet violate a constraint, but only yield undesired states later.

Punctuation constraints have the largest impact on coverage, reducing it to 93.2%. NP attachments, clauses, and modifier attachments also incur substantial coverage reductions. This suggests that ZPar's performance will degrade substantially over the sentences which it cannot cover, as they must contain constructions which are dispreferred by the model and fall out of the beam. Constraints with the smallest effect on coverage include root attachments, which only occur once per sentence and are rarely incorrect, and NP internal and PP attachments. For the latter two, the small error displacements suggests that alternate attachment points typically lie within the same projective span, limiting the changes required to attach them correctly.

Noun phrases

Applying NP attachment constraints causes a 4.4% drop in coverage for ZPar, and the effective constraint percentage is below 5% for both parsers. However, these constraints still result in the largest ΔUAS for both parsers, at 2.6% for MSTParser and 2.2% for ZPar. This reflects the prevalence of NP attachments in the corpus, despite the low error rates.

ΔUAS is split very evenly between correcting constrained arcs (1.4%) and cascaded arcs (1.2%) for MSTParser, while it skews more towards cascaded arcs for ZPar (1.0% and 1.4%). Most of the other error classes skew in the other direction. Repairing one NP attachment error typically repairs another non-NP attachment error, contributing to the large ΔUAS.

For NP internal attachments, both parsers have a similar error rate, with 206 effective constraints for MSTParser and 197 for ZPar. Although this class contains the second largest number of constraints, applying them gives the second smallest Δu for both parsers. This suggests that determining NP internal structure is a strength, even given the more complex OntoNotes 4 NP structure. Δc is also very small for both parsers, reinforcing the limited displacement and cascading impact of NP internal errors.

	NP attachment		NP internal	
Error class	MSTParser	ZPar	MSTParser	ZPar
NP attachment	-	-	45	69
NP internal	43	80	-	-
Modifier attachment	65	68	24	30
PP attachment	26	36	2	10
Coordination attachment	37	67	20	41
Clause attachment	59	65	1	1
Root attachment	24	21	2	1
Punctuation attachment	79	80	26	41
Other attachment	68	76	7	11
Total	401	493	127	204

Table 3.7: The number of cascaded errors repaired per error class when enforcing NP attachment and NP internal constraints for MSTParser and ZPar over WSJ 22.

Despite fewer effective constraints (i.e. less errors to fix), ZPar exhibits more cascading repair than MSTParser using both NP and NP internal constraints. This will be a common theme through this evaluation: the transition-based ZPar is better at propagating effective constraints into cascaded impact than the graph-based MSTParser, even though ZPar almost always begins with fewer effective constraints due to its better baseline performance. One possibility to explain this is that the beam is actually pruning away other erroneous states, while the graph-based MSTParser must still consider all of them.

Table 3.7 summarises error classes of the corrected cascaded arcs for the two NP constraint types. It is important to note that these constraints are closely related. NP attachment constraints directly identify the head of the NP as well as its correct attachment, providing strong cues for determining the internal structure. NP internal constraints implicitly identify the head of an NP. We can see that for both types of constraints, a large number of the cascaded corrections come from the other NP error class, illustrating this connection.

Table 3.7 also shows that, compared to MSTParser, ZPar repairs nearly twice as many NP internal and coordination errors when using NP attachment constraints,

and vice versa when using NP internal constraints. This suggests that ZPar has more difficulty identifying the correct heads for nominal coordination, and often chooses a word which should be a nominal modifier instead.

Coordination, Prepositional Phrases, and Modifiers

These three categories are large error classes for both parsers, with constraints leading to UAS improvements of 1.3 to 1.7% for each class in both parsers.

PPs and coordination have large effective constraint percentages relative to the other error classes for both parsers. However, they are also amongst the most isolated errors, with only 0.3% and 0.4% Δu for MSTParser and ZPar respectively across both classes. They also have a minimal impact on ZPar's coverage. This is surprising, and suggests that both classes have relatively contained attachment options within a limited projective span. The small error displacements for the classes reinforce this theory.

Modifiers are relatively isolated errors for MSTParser (0.5% Δu), but have slightly more cascading impact for ZPar (0.7% Δu). There are substantially more modifier constraints than PP or coordination, despite all of these yielding a similar UAS increase. This suggests that modifiers are actually relatively well analysed by both parsers, but there are so many of them that they are a relatively large source of error.

Clause attachments

MSTParser performs substantially worse than ZPar on clause attachments, with an effective constraint percentage of 17.9% compared to 13.0%, and Δc of 0.9% compared with 0.6%. MSTParser's error rate is the worst of any error class on clause attachments, while it is second to coordination attachments for ZPar. Attaching clauses is very challenging for dependency parsers, particularly considering the small size of the class.

ZPar again achieves a slightly larger cascaded impact than MSTParser (0.6% to 0.5%), despite having far fewer effective constraints. This suggests that the additional

clause errors being made by MSTParser are largely self-contained, as they have not triggered a corresponding increase in Δu.

Root attachment

Both parsers make few root attachment errors, though MSTParser is less accurate than ZPar. However, root constraints provide the largest UAS improvement per number of constraints for both parsers. Root errors are also the most displaced of any error class, at 9.3 words for MSTParser and 9.9 for ZPar. When the root is incorrect, it is often very far from its correct location, and causes substantial cascading errors.

Punctuation

Despite ignoring punctuation dependencies in evaluation, applying punctuation constraints led to substantial UAS improvements. On MSTParser, Δu is 0.1% (due to some punctuation not being excluded from evaluation), but Δc is 1.7%. On ZPar, the equivalent metrics are 0.2% and 1.5%. Enforcing correct punctuation has a disproportionate impact on the remainder of the parse.

For both parsers, punctuation errors occur more frequently than any other error type, with 469 and 430 effective constraints respectively (though the majority of these corrected errors are on non-evaluated arcs). ZPar's coverage is worst of all when enforcing punctuation constraints, suggesting that the remaining uncovered sentences will contain even more punctuation errors.

Incorrect punctuation heads are displaced from their correct locations by 7.4 words for MSTParser and 7.3 words for ZPar on average, second only to root attachments. Given that we are using projective parsers and a projective grammar, the large average displacement caused by errors suggests that punctuation affects and is in turn affected by the requirement for non-crossing arcs.

Table 3.8 summarises the error classes of the repaired cascaded arcs when punctuation constraints are applied. MSTParser has a more even distribution of repairs,

Error class	MSTParser	ZPar
NP attachment	75	51
NP internal	25	27
Modifier attachment	33	43
PP attachment	45	55
Coordination attachment	87	106
Clause attachment	66	48
Root attachment	59	27
Other attachment	65	69
Total	455	426

Table 3.8: The number of cascaded errors repaired per error category when enforcing punctuation constraints for MSTParser and ZPar over WSJ 22.

while ZPar's repairs are concentrated in coordination attachment. This suggests that MSTParser is relatively better at coordination as a proportion of its overall performance compared to ZPar. It also indicates that the majority of punctuation errors in both parsers (and especially ZPar) stem from incorrectly identified coordination markers such as commas.

Punctuation is commonly ignored in dependency parser evaluation (Yamada and Matsumoto, 2003; Buchholz and Marsi, 2006), and they are inconsistently treated across different grammars. Our results show that enforcing the correct punctuation attachments in a sentence has a substantial cascading impact, suggesting that punctuation errors are highly correlated with errors elsewhere in the analysis. Given the broad similarities between Stanford dependencies and other dependency schemes commonly used in parsing (Søgaard, 2013), we anticipate that the problems with roots and punctuation will carry across different treebanks and schemes.

Punctuation is often placed at phrasal boundaries and serves to split sentences into smaller sections within a projective parser. Graph-based and transition-based parsers, both of which use a limited local context to make parsing decisions, are equally prone to the cascading impact of erroneous punctuation. Influenced by poor punctuation parsing results, Ma et al. (2014) demonstrate how treating punctuation

Error class	cover	eff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
Baseline	100.0	-	-	-	83.1	77.4	-	-	-
NP attachment	92.3	1673	7.0	3.9	87.9	82.8	3.0	1.7	1.3
NP internal	96.7	1326	7.5	2.6	85.8	80.5	1.8	1.2	0.5
Modifier attachment	93.4	1860	14.7	3.8	87.2	82.6	2.6	1.8	0.7
PP attachment	96.9	1747	17.4	3.9	85.6	80.0	2.0	1.6	0.4
Coordination attachment	97.1	1968	26.8	5.8	86.1	80.9	2.4	1.8	0.6
Clause attachment	93.7	1172	22.5	6.2	85.9	80.9	1.9	1.2	0.7
Root attachment	98.5	1556	19.0	6.2	86.6	80.2	3.1	1.4	1.7
Punctuation attachment	84.0	2314	23.4	6.0	88.8	82.3	2.7	0.0	2.7
Other attachment	85.7	1673	12.7	4.2	89.4	85.3	2.9	1.8	1.1
All attachments	94.5	19426	17.4	4.8	100.0	100.0	15.6	15.6	0.0

Table 3.9: The coverage, effective constraints and percentage, error displacement, UAS, LAS, ΔUAS over the baseline, and the constrained and cascaded Δ for MSTParser over the sentences in the EWT development sections covered by ZPar.

tokens as properties of their neighbouring tokens rather than tokens in their own right can improve unlabeled dependency parsing accuracy. Other potential approaches could treat punctuation attachment as a global post-process, or incorporate more punctuation-specific features to try and account for its myriad roles in syntax could serve to improve performance.

3.8.2 Web Text

Tables 3.9 and 3.10 summarise the performance of applying constraints to MSTParser and ZPar over the combined development sections of the English Web Treebank. ZPar outperforms MSTParser by a slightly larger margin on web text, with the baseline systems scoring 83.7% and 83.1% UAS respectively. UAS falls by at least 8% for both parsers when run on web text, while the effective constraint percentage nearly doubles. A contributing factor is the POS tag quality — Petrov and McDonald (2012) found that many approaches to improving the parsing of web text focused on improvement POS tagging. Our MXPOST newswire POS tagger model scores only 90% accuracy on POS

Error class	cover	eff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
Baseline	100.0	-	-	-	83.7	79.4	-	-	-
NP attachment	92.3	1509	6.3	3.7	88.9	85.2	3.2	1.5	1.6
NP internal	96.7	1281	7.2	2.6	86.5	82.6	1.8	1.2	0.6
Modifier attachment	93.4	1659	13.1	3.6	87.8	84.7	2.5	1.6	0.8
PP attachment	96.9	1663	16.6	3.8	86.3	82.0	1.9	1.6	0.4
Coordination attachment	97.1	1774	24.2	5.7	86.7	82.3	2.2	1.7	0.5
Clause attachment	93.7	938	18.0	5.3	86.5	82.5	1.6	0.9	0.6
Root attachment	98.5	1512	18.5	6.5	87.2	82.2	3.0	1.4	1.7
Punctuation attachment	84.0	2024	20.5	5.9	89.7	84.8	2.6	0.0	2.6
Other attachment	85.7	1574	11.9	4.1	90.3	87.6	2.8	1.7	1.2
All attachments	94.5	18514	16.6	4.8	100.0	100.0	14.8	14.8	0.0

Table 3.10: The coverage, effective constraints and percentage, error displacement, UAS, LAS, ΔUAS over the baseline, and the constrained and cascaded Δ for ZPar over the EWT development sections.

tags over the combined Web Treebank development sections, compared to 96.3% over the WSJ section 22.

Error displacements are smaller across all error classes compared to newswire, though this is probably because of the shorter average sentence length in the Web Treebank. ZPar's coverage has also dropped across every error class. compared to newswire text, a sign of the increased divergence from the out-of-domain model and test texts. Punctuation attachment and Other attachment have the most impact on coverage, with reductions to 84.0% and 85.7% respectively. Both classes exhibit more complex behaviour in web text, though punctuation also recorded the lowest covered on newswire.

Noun phrases

NP attachment contributes 3.2% ΔUAS for ZPar and 3.0% for MSTParser, the largest and second largest improvements respectively despite both parsers exhibiting the lowest effective constraint percentage on this error class. Similar to newswire, ZPar's improvement skews towards cascaded arcs, while MSTParser's skews to constrained

arcs, though both parsers record a Δu greater than 1.3%. NP internal constraints exhibit much the same behaviour as they did on newswire, though both parsers make more than twice the number of errors on web text.

Coordination, Prepositional Phrases, Modifiers, and Clauses

Coordination is more prevalent in the Web Treebank than the WSJ, and it is the least accurate class for both MSTParser and ZPar with constraint percentages of 26.8% and 24.2% respectively. Overall ΔUAS improvements are 2.4% and 2.2%. Unlike newswire, ZPar markedly outperforms MSTParser. Coupled with the reduced number of NP internal constraints, it seems that the weakness of nominal coordinations in newswire is reduced (relatively) in web text for ZPar.

Conversely, PPs are rarer in web text than on newswire, and this class exhibits the smallest ΔUAS difference compared to newswire for both parsers. ZPar moves from 1.7% to 1.9% ΔUAS, and MSTParser from 1.8% to 2.0%, both of which are amongst the smallest over all error classes on web text. This is in spite of the effective constraint percentage being above 20% for both parsers. PPs are commonly incorrect, but infrequent enough to have less impact when they are incorrect on web text.

Aside from the increases in effective percentage and the Δ improvements over the baseline, modifier and clause attachments on web text perform very similarly to newswire. As on newswire, MSTParser performs substantially worse on clause attachment compared to ZPar, at 22.5% effective constraint percentage to 18.0%. The contribution of Δu and Δc to the overall UAS improvements, and the tendency for ZPar to extract more cascading impact from fewer effective constraints also continues to hold across these classes.

Root and Punctuation

All of the previous error classes on web text exhibited increased ΔUAS compared to their newswire equivalents. However, the bulk of these increases have come from Δc.

Δu is surprisingly static compared to newswire, increasing by at most 0.2% for both parsers, and only on classes with larger Δc values. Curiously, more effective constraints do not have a more pronounced cascading impact in web text.

By contrast, both the effective constraint percentage and Δu of root constraints rise substantially on web text. While ZPar was more accurate than MSTParser on newswire, the two systems are virtually tied on the Web Treebank. For both parsers, Δc is 1.4% and Δu is 1.7%, increases of over 1% from newswire. This is the highest ΔUAS for MSTParser, and second highest to NP attachment for ZPar.

Petrov and McDonald (2012) note that constructions such as imperatives, questions, and fragments are all more common in web text than newswire. All of these exhibit very different syntactic properties compared to newswire text, most especially in the movement of the root node closer to the front of the sentence. There are substantial projectivity implications of a sentence-initial root, as each arc emanating from the root divides the entire sentence into two disjoint components on either side. Nivre et al. (2014) found that enforcing the first word of a sentence as the root in a shift-reduce dependency parser dramatically improves the accuracy of parsing imperatives.

Applying punctuation constraints in web text also increases Δu for both parsers by over 1%, contributing to the largest cascading impact of any error class.

3.9 MSTParser at Full Coverage

Table 3.11 gives the breakdown of performance for MSTParser on all sentences in WSJ 22 and the Web Treebank development sections. These results are based on the same configuration as in Section 3.8.1, except we have not removed any sentences to match ZPar's coverage.

It can be seen that the parser makes more mistakes on all error classes, as expected. The most pronounced effects are on the two classes with the lowest coverage previously: other attachments, and punctuation attachments. 35.9% of punctuation tokens are

Error class	eff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
WSJ section 22								
baseline	-	-	-	91.3	87.5	-	-	-
NP attachment	388	5.7	5.0	93.8	90.4	2.6	1.4	1.2
NP internal	243	3.6	2.7	92.5	89.1	1.2	0.9	0.4
Modifier attachment	370	8.7	4.0	93.2	90.1	1.9	1.3	0.6
PP attachment	406	13.7	4.3	93.1	89.3	1.8	1.5	0.3
Coordination attachment	316	19.4	6.8	92.9	89.4	1.6	1.1	0.5
Clause attachment	266	19.5	7.2	92.9	89.5	1.7	1.0	0.7
Other attachment	311	9.4	6.6	93.2	90.4	1.9	1.1	0.8
Punctuation attachment	673	18.1	8.1	93.5	89.3	2.2	0.3	2.0
Root attachment	85	6.4	9.7	92.2	88.2	0.9	0.3	0.6
All attachments	3058	9.5	5.9	100.0	100.0	8.7	8.7	0.0
EWT development sections								
Baseline	-	-	-	83.1	77.4	-	-	-
NP attachment	2613	9.4	4.4	87.1	81.9	4.1	2.3	1.8
NP internal	1743	9.1	2.8	85.3	80.0	2.2	1.5	0.7
Modifier attachment	2739	18.5	4.2	86.5	81.9	3.5	2.4	1.1
PP attachment	2134	19.6	4.2	85.5	79.8	2.4	1.9	0.6
Coordination attachment	2543	30.7	6.4	86.1	80.8	3.0	2.2	0.8
Clause attachment	1836	28.2	7.3	85.7	80.7	2.7	1.6	1.0
Other attachment	4048	22.5	5.8	88.3	84.1	5.3	3.4	1.9
Punctuation attachment	5171	35.9	6.7	87.3	80.5	4.3	0.1	4.2
Root attachment	1671	20.1	6.5	86.4	79.9	3.3	1.5	1.8
All attachments	24498	19.1	5.5	100.0	100.0	16.9	16.9	0.0

Table 3.11: The effective constraints, difference to baseline, and percentage, error displacement, UAS, LAS, ΔUAS over the baseline, and the constrained and cascaded Δ for MSTParser over WSJ 22 and the concatenated EWT development sections.

incorrectly attached in the Web Treebank, while 30.7% of clauses and 28.2% of coordination are incorrect as well. The cascading impact of punctuation increases to 4.2% Δu, and the relatively infrequent punctuation error class records the largest overall ΔUAS from applying constraints, ahead of the far larger NP attachment class.

3.10 Summary

We have developed a procedure to classify the importance of errors in dependency parsers without any assumptions on how the parser will respond to attachment repairs. Our approach forces the parser to choose only correct arcs for certain words, whilst allowing it to otherwise form the highest scoring parse under its model. Compared to Kummerfeld et al. (2012), we can observe exactly how the parser responds to the parse repairs, though at the cost of requiring modifications to the parser itself.

Our results show that identifying the heads of NPs and their attachment points are challenges for both a graph-based and transition-based dependency parser. While both parsers correctly attach over 95% of NP in the WSJ section 22, and over 93% in the English Web Treebank development sections, correcting the remaining NP attachment errors contributes the largest total improvement to parsing accuracy of any of our error classes. This is due to the prevalence of NPs, as well as the substantial cascading impact of erroneously attaching an NP. Conversely, correcting NP internal structure, which implicitly specifies the NP head but not its attachment point, has amongst the smallest total impacts on parsing accuracy of all error classes, with little cascading effect elsewhere in the parse.

PPs, coordination, and modifiers are each large sources of error for both parsers. But unlike NP attachment, all of these error classes are relatively self-contained. Corrected arcs are rarely a long distance away from the parser's initial choice, limiting the cascading improvement, meaning that substantial, but not maximal gains in accuracy can be achieved by correcting each of these classes. Coordination and modifiers are

challenging on both newswire and web text, while PPs are more prevalent on newswire than web text, and decline in their contribution to accuracy improvements on web text relative to the other error classes.

ZPar outperforms MSTParser overall and across most error types, though MSTParser is relatively better at disambiguating nominal coordination. The most substantial difference between the two parsers comes on clause attachments, which are infrequent, but often erroneous in both newswire and web text. Beam search is capable of pruning most of the irrelevant states exhaustively searched by MSTParser without negatively affecting accuracy. However, beam search also causes ZPar to lose coverage when constraints are applied, while MSTParser can cover any sentence with consistent constraints. There is no principled way to address this whilst maintaining the original behaviour of the parser, though in practice, increasing the beam size and formulating a strategy to keep desired states in the beam can address the coverage reduction.

ZPar is also better than MSTParser at propagating the information from constraints into cascaded impact. Despite outperforming MSTParser, and thus having fewer errors to fix across most error classes, ZPar is still able to produce larger cascading contributions to the overall accuracy improvement from constraints. This may be due to the beam pruning erroneous states overall, avoiding the exhaustive consideration of the graph-based model in MSTParser.

Our results show that punctuation causes the largest cascading impact of all constraints across newswire and web text. Even when it is ignored in evaluation, incorrect punctuation serves as a strong signal for other errors in the sentence due to its usual placement at the edges of phrasal boundaries. Conversely, correctly attaching punctuation helps the parser to correct many other errors. More attention should be paid to punctuation in parsing, whether in the form of removing it entirely and re-adding it later, or addressing it more directly through features that capture its diverse behaviours.

We implement a robust procedure to identify the cascading impact of dependency parser errors. Our results provide insights into which errors are most damaging in

parsing, and will drive further improvements in parsing accuracy. In the next chapter, we develop features for MSTParser based on unannotated *n*-gram corpora which attempt to address some of the issues discussed in this chapter.

4 Surface and Syntactic *n*-gram Features for Dependency Parsing

In this chapter, we describe the implementation of *n*-gram-based features for MSTParser, encoding the frequency of surface *n*-grams and dependency subtrees in large unannotated text corpora. Such *n*-gram features have been shown to be effective in addressing some of the major sources of parser error discussed in the previous chapter, such as NP and PP attachment (Volk, 2001; Nakov and Hearst, 2005a). We use our constraint-based procedure to compare the performance of various feature types, investigating their effectiveness in addressing the issues we identified.

We use two large corpora extracted from scanned books: the Google Books Ngrams corpus (Michel et al., 2011), and the Google Syntactic Ngrams corpus (Goldberg and Orwant, 2013). The former is a collection of surface *n*-grams and their frequencies collected over 468 billion words of English scanned books. The latter is a collection of dependency subtrees and their frequencies collated over 345 billion words of scanned English books parsed with a shift-reduce parser and Stanford dependencies.

Features from surface *n*-gram counts over large web corpora such as Web1T (Brants and Franz, 2006) have proven to be useful syntactic hints (Bansal and Klein, 2011; Pitler, 2012a), but the impact of these features on out-of-domain text remains an open question. Additionally, prior feature analysis was restricted to examining error reductions based on POS tags of fillers. This is informative, but lacks the greater abstraction of our

dependency constraint error classes, and does not show whether the additional features also affected the cascading impact of each error class.

Surface *n*-gram features also rely on linear word order, and are unable to distinguish between syntactic and random co-occurrence. In contrast, longer *n*-grams are noisier and sparser, limiting potential features.

We develop features based on counts of syntactic *n*-grams from the Google corpus, trading off potential systemic parser errors for data that is aligned with the parsing task. Our work extends that of Bansal and Klein (2011, B&K), who found that surface *n*-gram features extracted over web text from Web1T frequencies were very effective in improving the accuracy of MSTParser. This success was in spite of the deficiency of surface *n*-grams for parsing. Additionally, we create second-order variants of B&K's surface *n*-gram features in parallel to second-order syntactic *n*-gram features, investigating whether the information captured in higher-order structures are useful for the parser.

We evaluate surface and syntactic *n*-gram features over newswire and web text across the LTH and Stanford dependency schemes. We also apply the constraint-based evaluation developed in Chapter 3 over our Stanford models, identifying which types of errors each feature set affects most, and whether they change the cascading impact of each error class.

Our results show that surface *n*-gram features using counts from Google Books Ngrams are largely equivalent to counts from Web1T across domains and formalisms. Syntactic *n*-gram features yielded slightly worse accuracy on in-domain text, but worked better on out-of-domain text across formalisms compared to surface *n*-grams, despite being better suited to the parsing task. Each type of *n*-gram feature also provided improvements for different error classes, consistent with the types of noise and error inherent in the *n*-gram source corpora.

Our best LTH system achieves 92.6% UAS on newswire and 85.9% UAS averaged over web text on a baseline of 92.0% and 84.6%. On Stanford dependencies, our

combined system achieves 92.1% on newswire and 84.6% averaged over web text over baselines of 91.3% and 83.1%. This combined system significantly outperforms each individual surface and syntactic *n*-gram feature type in isolation, demonstrating the complementary nature of these features.

Applying our constraint evaluation procedure shows that the combined system successfully reduces parser errors in the problematic error classes from the previous chapter — NPs, PPs, clause, and modifier attachments. The combined system is able to draw upon the strengths of both surface and syntactic features whilst avoiding their weaknesses. Surface *n*-gram features perform worse on coordination and verb attachments, whilst syntactic *n*-grams are weaker on nominal and PP attachments — structures that are known to be difficult to parse and hence more prone to erroneous subtree counts. For surface *n*-grams, Web1T features are better suited to PPs, and Google Books perform better on nominals.

Notably, the combined system is able to reduce the errors across most of our error classes, but these improvements are restricted to constrained impact, and do not markedly improve the cascaded impact of each class. This suggests that the features are addressing the most isolated errors, rather than those which cause additional issues in each sentence.

This is the first work to use features from the Syntactic Ngrams corpus in dependency parsing, and analyze the impact of surface and syntactic *n*-gram features across domains, parsing models, and dependency schemes. We hope that our results encourage further investigation into features from enormous parsed corpora, and demonstrate when they are effective.

This chapter is organised as follows. First, we discuss the *n*-gram corpora used in this work, and the literature on employing unannotated and automatically tagged resources to improve syntactic analysis. We describe our new syntactic *n*-gram features and the extraction process from the Google corpus. We discuss the B&K surface *n*-gram features, and our extensions to them. Finally, we present the results of applying and

combining different features from different datasets, and analyse the results using our constrained-based evaluation technique describe in Chapter 3, examining the impact of features across our error classes.

4.1 Background

4.1.1 *N*-gram Corpora

N-gram corpora collect *frequency counts* of small fragments of text from a very large text corpus. The *arity* of the *n*-gram is its size; for example, a *unigram* or *1-gram* contains one lexical unit, and a *trigram* or *3-gram* contains three. Importantly, these corpora are typically unannotated, and can be produced automatically through relatively simple techniques. However, it is usually necessary to collate these counts over very large collections of source documents, such as spidered websites or scanned books, in order to achieve high *n*-gram coverage and more reliable statistics. The resources required to create substantial *n*-gram corpora over web-scale text are prohibitive.

Despite the noise and lack of annotation, *n*-gram corpora can provide useful cues for many tasks in NLP. In this section, we describe the three corpora used in this thesis. Web1T and Google Books Ngrams are collections of *surface* n-*grams*, or plain word sequences. Google Syntactic Ngrams is a collection of *dependency parse subtrees*. These corpora were collected and produced by Google using their extensive data and computational resources. We also describe previous work with *n*-gram corpora for syntactic analysis.

4.1.1.1 Web1T

The Web1T corpus contains surface *n*-gram counts collated over approximately one trillion words of English web text taken from Google's web crawl (Brants and Franz, 2006). The counts represent the linear co-occurrence of words in web text. To produce the counts, the raw text was tokenised using similar rules to the Penn Treebank WSJ

```
hold a healing  150
hold a health   972
hold a healthy  536
hold a heap     249
hold a hear     179
hold a hearing  113241
hold a hearings 96
hold a heart    408
```

Figure 4.1: Raw 3-grams and their frequencies taken from Web1T.

data, and non-English encodings and junk-like text was discarded. Automatic sentence boundary detection was performed, and boundary markers <S> and </S> were inserted to mark the beginning and end of sentences. Counts were collected for all unigrams occurring more than 200 times in the source data, for *n*-grams of arity 2 to 5 occurring more than 40 times in the source data. The sentence boundary markers are included in the *n*-grams. All unigrams falling under the 200 frequency threshold were mapped to the token <UNK>. Figure 4.1 gives an excerpt from the 3-grams provided by Web1T.

Lin et al. (2010) and Pitler et al. (2010) document a number of noise issues in the source data of Web1T, which compromise the corpus integrity. Duplicate sentences, such as legal disclaimers and boilerplate text, occur millions of times in the source. Disproportionately short or long sentences, and primarily alphanumeric sentences are also likely to be garbage text. Lin et al. (2010) describe Google V2, which is an *n*-gram corpus developed from the same source documents as Web1T, but with additional preprocessing steps to address some of these noise sources. However, the V2 corpus has not seen widespread release as of writing, and so the original Web1T remains the most widely-used source of *n*-grams from web text.

4.1.1.2 Google Books Ngrams

The Google Books Ngrams English corpus (2012) contains counts of surface *n*-grams over 468 billion words sourced from scanned books published across three centuries (Michel et al., 2011). Like Web1T, Google Books collects *n*-grams of arity 1 to 5. Unlike

```
hold a battleship       82      44      29
hold a cornucopia_NOUN  75      58      40
hold a custom_NOUN      60      34      27
hold a hearing          27027   24988   16754
hold a rain             85      64      31
hold a rope_NOUN        985     713     498
hold a sling            43      26      14
hold a theology_NOUN    62      47      43
```

Figure 4.2: Raw 3-grams and their collated frequencies taken from Google Books Ngrams, including some *n*-grams with coarse POS tags.

Web1T, a uniform cutoff of 40 applies to all *n*-grams in this corpus, including unigrams, and any *n*-grams below this threshold were excluded completely. Google Books also leverages the temporal aspect of its source data, and provides a breakdown of *n*-gram frequencies by year of publication. Later efforts added coarse part-of-speech tags and syntactic labels to differentiate between different word senses in *n*-grams (Lin et al., 2012).

While Web1T was affected by the noise of web text, Google Books is affected by the accuracy of OCR and digitization tools, and the changing typography of books across time is one issue that may create spurious co-occurrences and counts (Lin et al., 2012).

4.1.1.3 Google Syntactic Ngrams

The Google Syntactic Ngrams English (2013) corpus[1] contains counts of dependency tree fragments over a parsed 345 billion word extract of the Google Books data (Goldberg and Orwant, 2013). The source is roughly one third the size of Web1T's, and three quarters the size of Google Books Ngrams.

The text was parsed using a beam-search shift-reduce parser based on ZPar (Zhang and Nivre, 2011) with the basic Stanford dependency scheme (de Marneffe and Manning, 2008b), henceforth the Goldberg and Orwant (2013) parser. Crucially, this parser and the first-order CRF POS tagger used for tagging the source data were trained over

[1] http://commondatastorage.googleapis.com/books/syntactic-ngrams

```
hold   hold/VBP/ROOT/0 a/DT/det/3 harp/NN/dobj/1      11   1915,1...
hold   hold/VBP/ROOT/0 a/DT/det/3 hat/NN/dobj/1       18   1873,3...
hold   hold/VBP/ROOT/0 a/DT/det/3 hatred/NN/dobj/1    33   1899,1...
hold   hold/VBP/ROOT/0 a/DT/det/3 head/NN/dobj/1      46   1794,1...
hold   hold/VBP/ROOT/0 a/DT/det/3 hearing/NN/dobj/1   174  1920,3...
hold   hold/VBP/ROOT/0 a/DT/det/3 heart/NN/dobj/1     49   1879,1...
hold   hold/VBP/ROOT/0 a/DT/det/3 heaven/NN/dobj/1    11   1825,1...
hold   hold/VBP/ROOT/0 a/DT/det/3 heresy/NN/dobj/1    49   1831,3...
hold   hold/VBP/ROOT/0 a/DT/det/3 heretic/NN/dobj/1   21   1829,1...
hold   hold/VBP/ROOT/0 a/DT/det/3 hide/NN/dobj/1      12   1884,1...
```

Figure 4.3: Syntactic 1-grams and their collated frequencies taken from the extended arcs set of Google Syntactic Ngrams. Each *n*-gram contains two content words, and the non-content determiner *a*. The fields are the head word, the *n*-gram, the total frequency, and the frequencies by year (truncated).

the union of the PTB WSJ section (Marcus et al., 1993), the Brown corpus (Kucera and Francis, 1967), and the Questions Treebank (Judge et al., 2006) — substantially more annotated data than typically used in dependency parsing. The use of so much extra training data suggests that parsing accuracy will be improved over typical systems trained only using the PTB. However, the Books source data is drawn from a different domain than the training data, which is known to reduce parsing performance. Our results from Chapter 3 show that ZPar trained on the OntoNotes 4 WSJ produces 8% lower unlabeled accuracy scores when parsing web text, though accuracies per each web domain varied substantially depending on how close the domain was to the original newswire training data.

The counts in Google Syntactic Ngrams distinguish between content and non-content words: non-content words are functional markers which serve to add polarity, modality, or definiteness, and are identified by specific dependency labels. Goldberg and Orwant (2013) note that the dependency labels *det*, *poss*, *neg*, *aux*, *auxpass*, *ps*, *mark*, *complm*, and *prt* identify non-content words, and that all of these (bar *poss*) are closed classes.

Corpus	Source	Source tokens
Web1T	Web text	1,000 billion
Google Books Ngrams	Scanned books	468 billion
Google Syntactic Ngrams	Parsed scanned books	345 billion

Table 4.1: Source text token counts for the *n*-gram corpora.

n-grams	Web1T	Books
1-grams	13,588,391	10,398,254
2-grams	314,843,401	142,086,111
3-grams	977,069,902	541,872,406
4-grams	1,313,818,354	824,306,624
5-grams	1,176,470,663	710,664,070

Table 4.2: A comparison of the *n*-gram distribution in the Web1T and Google Books corpora.

A syntactic *n*-gram contains n content words and $n-1$ arcs between those content words; the Syntactic Ngrams corpus provides counts over 2 to 5 content words (internally labeled as *arcs*, *biarcs*, *triarcs*, and *quadarcs* respectively). An *extended n*-gram also contains any non-content words which modify the content words; the corpus also provides counts over extended variants of the four types previously listed.

Figure 4.3 gives an excerpt from Google Syntactic Ngrams taken from the extended arcs dataset. The fields are the head word of the syntactic *n*-gram, the *n*-gram itself, the total *n*-gram frequency, and a frequency breakdown by year. The *n*-gram format is word/POS/label/head with a 1-based indexing scheme; the word with a 0 head index is the root of the subtree. The words in the *n*-gram maintain their original sentence ordering, but intervening words without a dependency arc to any word in the *n*-gram are omitted (Goldberg and Orwant, 2013).

Each subtree in the Google Syntactic Ngrams corpus must appear at least 10 times, a much lower cutoff than either of the surface *n*-gram corpora used in this work. Subtrees below this cutoff are not included in the corpus.

Table 4.1 lists the source size and type of each corpus, and Table 4.2 gives a breakdown of the unique surface *n*-grams of each size appearing in the Web1T and Google

Books Ngrams corpus. It can be seen that despite the lower unigram frequency cutoff, Web1T contains more distinct *n*-grams of every arity than Google Books. This reflects the larger source corpus used for Web1T, as well as the vocabulary expansion inherent in web text due to URLs, emoticons, and jargon.

4.1.2 Using *n*-grams for Syntax

The use of automatically parsed data for improving parser accuracy has been explored in several different ways. Sarkar (2001) and Steedman et al. (2003) have applied *co-training* techniques to constituency parsing. In co-training, two systems are iteratively trained on a small amount of annotated data augmented with the most confident predictions of the other system, allowing each to benefit from the insights of the other. Both works found that co-training is best suited to situations where there is only a small amount of labeled data available. Sagae and Tsujii (2007) have also found co-training to be useful for domain adaptation, using parsed out-of-domain data to augment annotated training data.

Volk (2001) addressed PP attachment ambiguities in a 3,000 sentence German corpus by comparing the frequency of hits of different configurations of preposition, attachment point, and noun from a web search API. The raw hit counts are little more than an indicator of surface-form affinity, but they contribute to over a 10% improvement in resolving PP attachments compared to simply setting the attachment point to the most frequently observed.

Web search hit counts were also used by Lapata and Keller (2004) as cues for determining noun compound bracketing. They compute a probability based on the ratio of web page hit counts for query terms based on the fragment implied by competing brackets. The most likely bracketing under this model is chosen and returned. Using only these *n*-gram statistics, their system achieved an accuracy of 78.68%, competitive with a state-of-the-art system that used a large predefined taxonomy of bracketings and thesaurus.

Nakov and Hearst (2005a) further improved noun compound bracketing accuracy to 89.34% on a larger version of the same task (using twice the test data) by greatly expanding the number of web *n*-gram count features. In particular, they introduced *paraphrase-style* features, where the query term is rewritten using a fixed set of static rewritings. For example, for the nominal compound *brain stem cells*, two competing paraphrases are *stem cells found in the brain* (right branching), or *cells found in the brain stem* (left branching). High hit counts for the paraphrases associated with a particular branching provide additional evidence for that branching. Nakov and Hearst (2005b) extended these features to resolving PP and nominal coordination attachments in English, achieving over 80% precision for both tasks.

Bansal and Klein (2011) adopted the idea of *n*-gram frequency statistics for full-scale parsing. By this time, web search engines were introducing increasing limits to their APIs, preventing *n*-gram statistics of individual query terms from being collected on a wide-scale. The quadratic number of possible attachments in parsing also presented scaling problems with querying systems.

Instead, they used Web1T as a static, offline source of web *n*-gram counts for parsing feature. They represented competing attachments as surface *n*-grams of the attachment point and word in their sentence order, and calculated the bucketed frequencies of these *n*-grams efficiently by searching Web1T. They also used Nakov and Hearst (2005a)'s paraphrase-style features, but rather than employing a fixed number of paraphrase patterns, they mined all possible patterns using the Web1T 3-grams. Their work resulted in a 0.6% ΔUAS improvement baseline MSTParser to 92.0%, and a 1.2% improvement in PARSEVAL F-score over the baseline Berkeley reranker. We use Bansal and Klein (2011) as the basis for our new syntactic *n*-gram features, and it will be discussed further throughout the remainder of this chapter.

Chen et al. (2009) achieves substantial parsing accuracy improvements in English and Chinese using subtree-based features. They parse the 30 million word BLLIP corpus using the first-order MSTParser, and count the frequency of subtrees containing

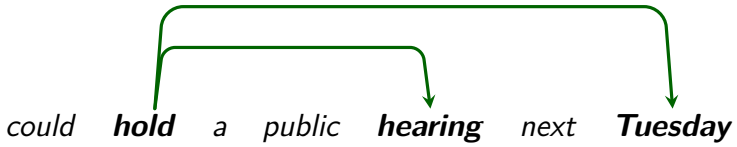
up to 3 arcs. The subtree frequencies are coarsely bucketed, with new indicator features developed for MSTParser that are fired when a stored subtree is encountered in the parsing process. They achieve a 0.6% ΔUAS improvement to 92.5% on Penn2Malt WSJ section 23 dependencies. Combining their features with the word cluster features of Koo et al. (2008) yields a 1.4% increase to 93.2%, competitive with the state-of-the-art. This work creates features that are similar in design to our syntactic *n*-gram features. However, Chen et al. (2009) employ the same underlying parser and dependency formalism to generate their subtree counts as they do for final evaluation. The counts are also extracted over an in-domain newswire corpus that is orders of magnitude smaller than Google Syntactic Ngrams.

Other recent work on improving dependency parsing using *n*-gram features include Pitler (2012a), who exploits Brown clusters and point-wise mutual information of surface *n*-gram counts to specifically address PP and coordination errors. Chen et al. (2013) describe a novel way of generating meta-features by parsing a large corpus, and counting the number of activations for each feature during parsing. The relative bucketed activation frequencies are encoded with the original features to produce the meta-features, emphasising the important feature types used by the parser.

4.2 Features in MSTParser

MSTParser factors its parsing decisions over individual and pairs of dependency arcs originating from a common parent. *First-order* features may be defined over a head and argument word pair, with access to the direction of the proposed arc, distance between the words, the POS tags, and any other lexical information provided to the parser for those words. *Second-order* features may be defined over a head and two of its arguments, where both of the arguments must be immediately adjacent to each other with respect to all of the arguments of the head. The arguments must also both be on the same side of the head. *Triple* features refer to the head and two arguments together,

while *sibling* features refer only the two adjacent arguments The side on which the arguments are relative to the head, the relative position of all three items, POS tags and any other lexical information supplied to the parser are available for use in these features. The restricted nature of the second-order features is due to the way they are implemented within Eisner's algorithm (McDonald and Pereira, 2006). Parse 4.1 gives a visual depiction of the second-order factorisation.



Parse 4.1: The second-order factorisation used in MSTParser, with a head and two adjacent arguments, both on the same side of the head.

4.3 Surface *n*-gram Features

B&K demonstrate that features generated from bucketing simple surface *n*-gram counts over Web1T are useful for almost all attachment decisions. However, this technique is restricted to counts based purely on the linear order of adjacent word, and is unable to incorporate disambiguating information such as POS tags to avoid spurious counts. B&K also tested only on in-domain text, though these external count features should be useful out of domain as well.

In this section, we describe the B&K first-order surface *n*-gram features, and present our second-order extension, attempting to capture deeper syntactic relationships from the co-occurrence statistics. We test the effectiveness of these features when extracted from Web1T and Google Books Ngrams. Importantly, the same underlying features are used in our comparison; the only difference is the source of the bucketed counts encoded in the features.

4.3.1 First-order surface *n*-gram features

Affinity features rely on the intuition that frequently co-occurring words in large unlabeled text collections are likely to be in a syntactic relationship (Nakov and Hearst, 2005a). Given a proposed dependency arc, the head and argument may be combined to form a contiguous query n-gram, with the first word decided by the directionality of the proposed arc. As both Web1T and Google Books Ngrams contain surface n-grams of up to length 5, additional query n-grams may be formed by allowing up to three intervening word between the head and argument. The total sum of the counts of these query n-grams can be extracted from Web1T and Google Books, giving a frequency-based cue for the likelihood of the proposed arc.

The raw frequencies extracted from the counts corpus are too sparse for direct use as features. Instead, B&K use the following equation to discretise the count into a small number of buckets that are encoded in the feature:

$$\text{bucket} = \left\lfloor \frac{\log_2(\text{count})}{5} \right\rfloor \tag{4.1}$$

Each surface n-gram feature includes the POS tags of the proposed head and argument, the discretised count, dependency direction, and binned length as well as the bucket. Additional cumulative features are generated with each bucket up to the maximum bucket value observed in the training data to address sparsity in the data; features with lower frequencies in the n-gram corpus will receive additional features through this.

B&K also extend the paraphrase-style features of Nakov and Hearst (2005a). Instead of static patterns, they generate these features automatically using n-gram frequencies. For each proposed dependency, 3-grams of the form ($\star$ q_1 q_2), (q_1 $\star$ q_2), and (q_1 q_2 $\star$) are extracted, where q_1 and q_2 are the head and argument in their linear order of appearance in the original sentence, and $\star$ is any single context word appearing before, in between, or after the query words respectively. The most frequent words appearing

in each of these configurations for each proposed dependency are encoded as features with the head and argument POS tags.[2]

For example, given the arc *hold* → *hearing* in Parse 4.1, *public* is the most frequent word appearing in the surface *n*-gram (*hold* ⋆ *hearing*) in Web1T. Thus, a mid-word paraphrase feature POS (*hold*) ∧ POS (*hearing*) ∧ *public* ∧ `mid` would be encoded for this arc. The POS tags are used in the final feature to allow for generalisation beyond the original words, while the context word itself mimics the static words in the templates used by Nakov and Hearst (2005a).

A further generalisation substitutes the context word *public* with its POS tag, as determined by a simple unigram POS tagger trained on the WSJ training data.

The overhead of looking up surface *n*-gram queries is mitigated in our research setting by pre-computing all possible dependency arcs in the training and test data, which is simply all directed pairs of words in every sentence. Then, the *n*-grams are extracted ahead of parsing and cached in lookup tables. These tables are loaded by the parser at runtime, necessitating an in-memory query for each constructed *n*-gram for its count. This is not a general solution in practice, but is sufficient to validate our features. However, it is very memory intensive, requiring lookup tables to store hundreds of thousands of arcs and their counts. Techniques such as feature hashing and pruning to reduce the feature space would be useful in alleviating the memory pressure for real-world use.

4.3.2 Second-order surface *n*-gram features

In this section, we describe how we extend B&K's first-order features to second-order. Reflecting the factorisation used in MSTParser, our second-order structures are triples and siblings, where a triple is a head with two of its adjacent arguments on one side, and siblings are the two arguments in a triple, excluding the head.

[2]The top 20 words in between and top 5 words before and after each query *n*-gram are used by Bansal and Klein (2011), and we follow these settings in this thesis.

In Parse 4.1, there is a second-order structure involving *hold*, *hearing*, and *Tuesday*. We extract a triple n-gram containing all three words in their sentence order, and a sibling n-gram of only the argument words. We then scan the n-gram corpus for each possible configuration (including intervening words) that the query n-gram may appear in; for triples, these are:

- $(q_1\ q_2\ q_3)$
- $(q_1\ \star\ q_2\ q_3)$
- $(q_1\ q_2\ \star\ q_3)$
- $(q_1\ \star\ q_2\ \star\ q_3)$
- $(q_1\ \star\ \star\ q_2\ q_3)$
- $(q_1\ q_2\ \star\ \star\ q_3)$

where q_1, q_2, and q_3 are the words of the triple in their linear order, and $\star$ is a single intervening word of any kind.

A limitation of higher-order surface n-gram features from a corpus of arity 5 is that the maximum number of intervening words is reduced to two. This constricts the range of these features compared to first-order, and prevents triple and siblings structures extending over longer distances from receiving appropriate counts.

Once we have extracted the counts for triple structures, we sum and bucket them as per Equation 4.1. The final feature names for the proposed second-order arcs include the properties as the first-order features, along with the bucket and the binned distance between the two children. As with first-order features, the query n-grams are pre-computed and their counts pre-extracted and cached to avoid the overhead of scanning the n-gram corpus during runtime.

Sibling structures only involve two words, and so we extract surface affinity and paraphrase n-gram features for them as per Section 4.3.1.

4.4 Syntactic *n*-gram Features

Unlike surface n-grams, syntactic n-grams are not restricted to linear word order and are not affected by random co-occurrence. Attachment decisions in parsing can commonly

span over distances longer than three intervening words, which is the maximum distance which the Web1T and Google Books Ngrams corpora can support. Using subtree fragments avoids the length limitations, as the subtree will naturally extend over as many words as is necessary. The trade-off is that syntactic *n*-grams are generated with a parser over out-of-domain text, so there are undoubtedly systemic parser errors. The question is whether the scale of the corpus and the closer fit to the parsing task outweigh the simpler word affinity available in surface *n*-gram corpora and the introduction of parser error.

In this section we describe our first- and second-order syntactic *n*-gram features, and show how they are extracted and applied in MSTParser.

4.4.1 First-order syntactic *n*-gram features

Our first-order syntactic *n*-gram features reinforce high-frequency dependencies identified in the corpus. Given a proposed dependency arc, we search the arcs component of the Syntactic Ngrams corpus for subtrees which match it, summing their counts. The total count is discretised, and encoded as a feature name for the arc, along with the parameters used to find the count. Importantly, we search for unlabeled arcs, and do not use the label for any features. This is done for two reasons: first, because MSTParser assigns labels in a separate subroutine to the decoding of the most likely arc, and second, some of our experiments use a different dependency scheme to the Syntactic Ngrams corpus.

As Syntactic Ngrams contains POS tag and directionality information in its subtrees, we can extract many more types of features based on different search parameterisations. We take different combinations of the words, POS tags, and directionality of the proposed arc, and extract counts for each combination. For example, we can search for all subtrees which:

- match the proposed arc exactly, including the relative ordering of head and argument;

Feature Lookup	Count	Bucket
hold (head)	80,129k	4
hearing (arg)	7,839k	4
hold → *hearing*	15k	3
hold → *hearing* (head at left)	15k	3
VB (head)	20,996,911k	5
NN (arg)	22,163,825k	5
VB (child at right)	6,261,484k	5
NN (head at left)	15,478,472k	5
VB → NN	1,784,891k	5
VB → NN (head at left)	1,437,932k	5
hold → NN	7,362k	4
hold → NN (head at left)	6,248k	4
VB → *hearing*	396k	3
VB → *hearing* (head at left)	354k	3

Table 4.3: First-order syntactic *n*-gram features, their counts in the extended arcs dataset, and the bucketed count for the *hold*→*hearing* arc in Parse 4.3.

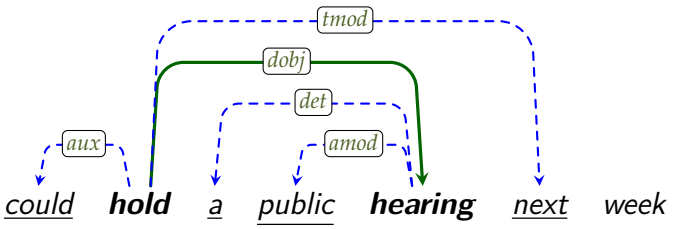
- match the proposed arc, ignoring the relative ordering of head and argument;
- have the same POS tags for the head and argument as the proposed dependency;
- are headed by the head word of the proposed arc, but have any argument word;
- are headed by the head POS of the proposed arc, but have any head word.

This spectrum of additional features allows us to abstract away from the reliance on the relative word-order of the head and argument, which is a critical limitation of using surface *n*-gram features.

Table 4.3 summarises the individual first-order features extracted from the Syntactic Ngrams corpus given the proposed dependency arc *hold* → *hearing* depicted in Parse 4.2. Each feature is listed with its count in the extended arcs dataset along with computed bucket value that will be encoded as a representation of its frequency. The bucket value is calculated using Equation 4.1 as per the surface *n*-gram features.

The feature combines the head and argument POS tags, the dependency direction, binned length, and bucket. We again encoded cumulative features for each bucket up to the maximum value observed in the training data.

We also experiment with paraphrase-style features extracted from the extended biarcs dataset. Unlike surface n-gram paraphrase features, which assume that any word appearing before, in between, or after a query n-gram is a potential paraphrase, the Syntactic Ngrams corpus allows us to only choose paraphrase targets which have a direct syntactic link to either the head or argument word. We can also use the POS tag of the paraphrase words directly as these tags are included in the Syntactic Ngrams corpus, rather than use the output of the unigram tagger necessary for surface n-grams.



Parse 4.2: The paraphrase-style context words around *hold*→*hearing* in a syntactic n-gram. Context words are underlined and their arcs dashed.

Parse 4.2 depicts a sample subtree centred on a dependency arc *hold* → *hearing*. The potential context words are underlined; these are the words which are connected by a dependency arc to either the head or argument word. Given the proposed dependency arc, we search the biarcs Syntactic Ngrams data for all context words connected to the proposed arc, and sum their counts. The most frequent words before, in between, and after the query n-gram are encoded as features in the same manner as the first-order surface n-gram paraphrase features described in Section 4.3.1.

We reduce the cost of searching for matching subtrees by using the same preprocessing procedure as surface n-grams.

4.4.2 Second-order syntactic *n*-gram features

We experiment with a number of second-order features, mirroring those extracted for triple and sibling structures and surface *n*-grams in Section 4.3.2. Analogous to the first-order case, we can extract subtrees from the Syntactic Ngrams biarcs corpus only when they precisely match the proposed triple and sibling structures, rather than relying on linear word co-occurrence.

Given a sibling or triple structure, we search the Syntactic Ngrams corpus for subtrees which match exactly, either for all three words, or all three POS tags. The search is subject to the requirement that the relative order of the head and arguments to each other is maintained. We sum the counts of all matching subtrees, and bucket the sum as per Equation 4.1. The final generated features are analogous to Section 4.3.2, with the only difference being the source of the counts.

The second-order syntactic *n*-gram features subsume some of the paraphrase-style features described in Section 4.4.1, in that some of the context words extracted by the paraphrase process will overlap with the triple and sibling structures. We discuss the effectiveness of the features in isolation and together in our results.

4.5 Experimental Setup

We implement our features using MSTParser, using the same configuration as described in Chapter 3. We briefly attempted to implement the surface and syntactic *n*-gram features described in this chapter for the transition-based ZPar system, which we used for our constraints evaluation procedure in Chapter 3. Our intention was to perform a cross-parser evaluation, comparing the impact of the features under a different parsing algorithm. However, we were unable to achieve any substantial improvements over the baseline parser with any combination of *n*-gram features. As ZPar also does not support the LTH formalism, we did not continue to pursue this line of research, and note it as a point of future work.

We evaluate using UAS over non-punctuation dependencies using the same evaluation script as described in Section 3.5.1. We omit an extensive discussion of labeled attachment scores in this thesis for brevity, but they are consistent with the reported UAS scores. Statistical significance is computed at $p < 0.05$ using Dan Bikel's stratified shuffling script and `eval06.pl`.[3]

4.5.1 Dependency schemes

LTH Dependencies

We used LTH dependencies generated over the Vadas and Curran (2007) NP-enriched PTB, following Bansal and Klein (2011) and Pitler (2012a). We also followed the evaluation procedure of the SANCL 2012 shared task using the answers, newsgroups, and reviews sections of the English Web Treebank (described in Chapter 2) for out-of-domain evaluation. Each section was converted to LTH dependencies with pennconverter in a similar fashion to the WSJ data.

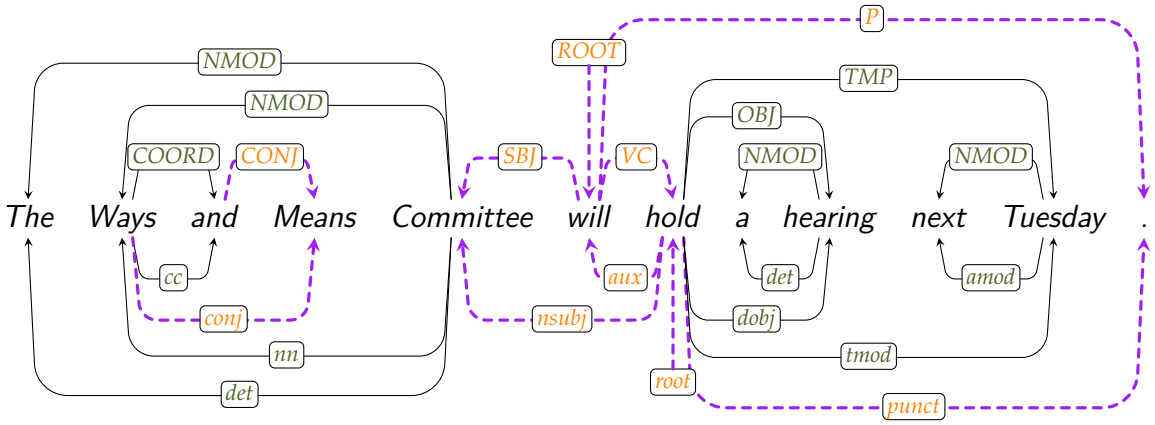
WSJ section 22 and the Web Treebank development sections were used as development data, and ran the best performing configurations as a final test over WSJ section 23 and the Web Treebank test sections. The additional Web Treebank evaluation allows us to test B&K's original surface *n*-gram features and our own on an out-of-domain corpus.

We generated POS tags for the data using MXPOST (Ratnaparkhi, 1996) in an identical fashion as described in Section 3.5.1.

Stanford Dependencies

To test the effectiveness of our features across different dependency formalisms, we also trained and evaluated MSTParser over the Stanford dependency configuration described in Section 3.5.1. This involved a change in the WSJ corpus from the Penn Treebank to the OntoNotes 4 release; we used the POS tagging setup described in

[3] Available at http://ilk.uvt.nl/conll/software.html#eval



Parse 4.3: LTH (top) and Stanford (bottom) dependencies. Dashed arcs indicate unlabeled differences between the schemes; solid arcs are isomorphic between both trees.

Section 3.5.1 to ensure consistent tags. There was no change in the Web Treebank source data, other than using the original Stanford dependency release. All other components of the evaluation procedure remained the same between the two schemes.

Stanford dependencies were used to produce the Google Syntactic Ngrams corpus, so the evaluation on this scheme allows us to investigate whether the cross-dependency scheme nature of the syntactic *n*-gram features helps or hinders performance. It also allows us to judge the effectiveness of B&K's original surface *n*-gram features and our extensions to them across formalisms.

All features for each dependency scheme are generated and cached independently due to the different training and test corpora.

Comparing LTH and Stanford Dependencies

Parse 4.3 gives an LTH and Stanford dependency analysis of a simplified sentence from the PTB section 22. The most notable differences are the treatment of auxiliary or main verbs as roots of the sentence (and the cascading attachment differences from the root decision), and coordination attachment to the coordinator or the coordinated word. In the LTH scheme, sentences containing an auxiliary verb (*will* in this case) are headed by that auxiliary, whilst the Stanford scheme treats the main predicate as the

	Count	%
Dependencies in common	707,812	74.5
Dependencies different	242,216	25.5
Total	950,028	100.0

Table 4.4: Common and different unlabeled dependencies between the LTH and Stanford schemes over PTB sections 02-21 with Vadas and Curran (2007)'s NP brackets.

head and the auxiliary as a dependent of it. This distinction has a cascading effect on many other dependency types which attach to the head of a sentence, including punctuation, subject and object dependencies. Coordination is another important source of difference; in the LTH scheme, there is no direct link between coordinated items, whilst the Stanford scheme explicitly links them together under a *conj* label.

Table 4.4 gives the summed total of common and different unlabeled dependencies between the LTH and Stanford schemes, as computed over PTB sections 02-21, enriched with the Vadas and Curran (2007) NP brackets. Nearly 75% of the dependencies have the same head and argument, with most differences stemming from the cascading impact of auxiliaries as roots. Coincidentally, this proportion almost exactly matches the observed differences in Parse 4.3.

Søgaard (2013) conducts a detailed comparison of these schemes, concluding that many differences are offset by the biases of parsers; comparing parser output across schemes leads to higher correlation than comparing gold-standard dependencies. They also find that LTH and Stanford dependencies are similarly useful for a variety of downstream applications, though LTH is notably worse at sentence compression due to its representation of coordination.

	WSJ 22			EWT Dev		
LTH	WEB1T	BOOKS	SYN	WEB1T	BOOKS	SYN
Baseline	92.3	92.3	92.3	83.7	83.7	83.7
+1st	-	-	92.8	-	-	84.5
+1st +para	92.8	92.8	92.8	84.6	84.5	84.4
+1st +2nd	-	-	**92.9**	-	-	84.5
+1st +para +2nd	**93.0**	**92.9**	92.7	84.6	84.6	84.4

Table 4.5: LTH UAS on WSJ 22 and the macro-averaged English Web Treebank answers, newsgroups, and reviews development sections for accumulative Web1T and Google Books surface *n*-grams and syntactic *n*-grams features. All results are significant improvements over the baseline, and bolded results are significant improvements over the column.

4.6 Results

In this section, we discuss our results, including the performance of different first- and second-order feature types on the development sets, comparisons between surface and syntactic *n*-gram features and different sources of surface *n*-grams, and the performance of a system combining the best individual feature sets.

Tables 4.5 and 4.6 summarise the results of using different combinations of first- and second-order surface and syntactic *n*-gram features over the WSJ section 22 and English Web Treebank development sets. We tested two configurations of surface *n*-gram features: the affinity and paraphrase features defined by Bansal and Klein (2011), and a combination of those features with our additional second-order surface *n*-gram features. We tested four configurations of syntactic *n*-gram features, adding a permutation over the paraphrase features.

All of our feature types were significant improvements over the baseline parser in and out-of-domain. On LTH dependencies, the best performing feature combination from each *n*-gram source outperformed the baseline by 0.6 – 0.7% UAS, reaching a best score of 93.0%. Web text improvements were 0.8 – 0.9% UAS, reaching 84.6% over a baseline of 83.7%. On Stanford dependencies, surface *n*-gram features outperformed

	WSJ 22			EWT Dev		
Stanford	WEB1T	BOOKS	SYN	WEB1T	BOOKS	SYN
Baseline	91.3	91.3	91.3	82.5	82.5	82.5
+1st	-	-	91.7	-	-	83.6
+1st +para	91.8	92.0	91.6	83.4	83.5	83.4
+1st +2nd	-	-	**91.8**	-	-	83.7
+1st +para +2nd	**92.0**	92.0	-	83.5	83.6	-

Table 4.6: Stanford UAS on WSJ 22 and the macro-averaged English Web Treebank answers, newsgroups, and reviews development sections for Web1T and Google Books surface *n*-grams and syntactic *n*-grams features. All results are significant improvements over the baseline, and bolded results are significant improvements over the column.

syntactic *n*-gram features on newswire, with +0.7% ΔUAS compared to +0.5% ΔUAS. However, syntactic *n*-gram features performed slightly better out-of-domain, with a +1.2% ΔUAS over the baseline to 83.7%, compared to 83.5% for Web1T surface *n*-grams and 83.6% for Google Books surface *n*-grams.

4.6.1 Surface *n*-gram Features

We found that second-order features slightly improved UAS for both LTH and Stanford dependencies and both corpora, and did not reduce UAS for any configuration on top of first-order and paraphrase features.

There was very little difference between choosing Web1T or Google Books as a source of *n*-grams in these experiments. Web1T *n*-grams performed slightly better for LTH dependencies, while Google Books Ngrams performed slightly better for Stanford dependencies. Most configurations saw no statistically significant differences between the two.

LTH	BASE	WEB1T	BOOKS	SYN	WEB1T SYN	Δ
WSJ 23	92.0	92.4	92.5	92.4	**92.6**	+0.6
EWT Answers	83.5	84.3	84.0	84.5	**84.8**	+1.3
EWT Newsgroups	86.1	86.8	87.0	87.1	**87.4**	+1.3
EWT Reviews	84.3	85.0	84.8	85.3	**85.7**	+1.4
EWT Average	84.6	85.3	85.3	85.7	**85.9**	+1.3

Table 4.7: LTH UAS on the WSJ and English Web Treebank (EWT) answers, newsgroups, and reviews test corpora for the baseline, Web1T, Google Books, Syntactic, and combined feature sets. All results are statistically significant improvements over the baseline.

4.6.2 Syntactic *n*-gram Features

On LTH, there is no significant difference between between first-order surface and syntactic *n*-gram features. For Stanford, syntactic *n*-gram features performed significantly worse than surface *n*-grams, particularly when compared to Google Books Ngrams features.

When adding second-order features to first-order features, UAS improves slightly across both schemes and domains. However, adding syntactic paraphrase features did not improve accuracy over first- and second-order features, and reduced it in some cases. Restricting paraphrase features to words only over syntactic links seems to reduce their effectiveness — an interesting result, as Bansal and Klein (2011) found these features were amongst the most informative for surface *n*-grams.

4.6.3 Combining Surface and Syntactic *n*-gram Features

Tables 4.7 and 4.8 summarise our best test set results for LTH and Stanford dependencies respectively, using the best performing features from the development sets: first-order, paraphrase, and second-order features for surface *n*-grams, and first-order and second-order features for syntactic *n*-grams. We also test a combined model of the best performing surface and syntactic *n*-gram features together; on LTH dependencies

Stanford	BASE	WEB1T	BOOKS	SYN	BOOKS SYN	Δ
WSJ 23	91.3	92.0	92.0	91.8	**92.1**	+0.8
EWT Answers	81.5	82.5	82.3	82.7	**83.0**	+1.5
EWT Newsgroups	85.3	86.6	86.4	86.7	**87.1**	+1.8
EWT Reviews	82.4	83.1	83.1	83.6	**84.1**	+1.7
EWT Average	83.1	84.1	83.9	84.4	**84.7**	+1.6

Table 4.8: Stanford UAS on the WSJ and English Web Treebank (EWT) answers, newsgroups, and reviews test corpora for the baseline (BASE), Web1T (WEB1T), Google Books (BOOKS), Syntactic (SYN), and combined (BOOKS + SYN) feature sets. All results are statistically significant improvements over the baseline.

		EWT			
System	WSJ 23	ANS	NGS	REV	Average
SANCL 2012 Baseline	91.5	81.6	85.2	83.3	83.4
Pitler (2012b)	92.0	82.3	86.1	82.9	83.8
McClosky et al. (2012)	92.0	82.6	**87.2**	**84.4**	**84.7**
Combined *n*-gram features	**92.1**	**83.0**	87.1	84.1	**84.7**

Table 4.9: The comparison of our system against systems from the SANCL 2012 shared task.

this combined surface features from Web1T, while on Stanford surface features from Google Books Ngrams were used.

The development results are consistent on the test set. Each feature type in isolation provides a significant 0.4 – 0.5% UAS improvement over the baseline parser on newswire, and 0.5 – 1.0% on web text. For LTH dependencies, the combined model achieves +0.6% ΔUAS over the baseline on newswire, and +1.3% averaged across the web text domains. For Stanford dependencies, the combined system achieves +0.8% ΔUAS on newswire and +1.6% on web text, significantly outperforming all other types in isolation. These results illustrate the complementary nature of surface and syntactic *n*-gram features.

Table 4.9 compares our combined system against the baseline and top performing non-stacked dependency parsers from the SANCL 2012 shared task. We outperform

all of these systems on newswire and the answers section of the Web Treebank, whilst equalling the best average across the three web text domains. However, our combined features and the non-stacked participants are well behind the best results in the task, which used combinations of several constituency and dependency parsers together with a variety of voting, bagging, and merging schemes. The best dependency parsing combination also dramatically improved the POS tagging accuracy over web text (Petrov and McDonald, 2012), contributing to its large accuracy advantage (Zhang et al., 2012).

While our syntactic *n*-gram counts are computed using the Stanford dependency scheme, the SYN column in Tables 4.7 and 4.8 show these features produced roughly equivalent absolute UAS improvements in- and out-of-domain on both schemes. This presents further evidence for the Søgaard (2013) hypothesis of parser bias smoothing differences between schemes.

4.7 Analysis

We analyse performance on Stanford dependencies, as our constraint-based evaluation developed in Chapter 3 uses this scheme. However, our observations are broadly applicable across both dependency schemes.

Figure 4.4 gives an error breakdown over WSJ 22 by high-frequency filler gold POS tag on Stanford dependencies. The baseline, Web1T surface *n*-grams, syntactic *n*-grams, and combined systems reported in Table 4.8 are compared in the table. We can see that for most tags, the combined system outperforms the baseline and makes equal or fewer errors than either the surface or syntactic *n*-gram features in isolation. We can also see that the combined system is able to overcome situations where one of the feature sets performs poorly, such as prepositions. Notably, the combined system has little impact on coordination, where all feature types are weak.

Syntactic *n*-gram features are outperformed by surface *n*-gram features on Stanford dependencies. We can see that prepositional attachments and some noun attachments

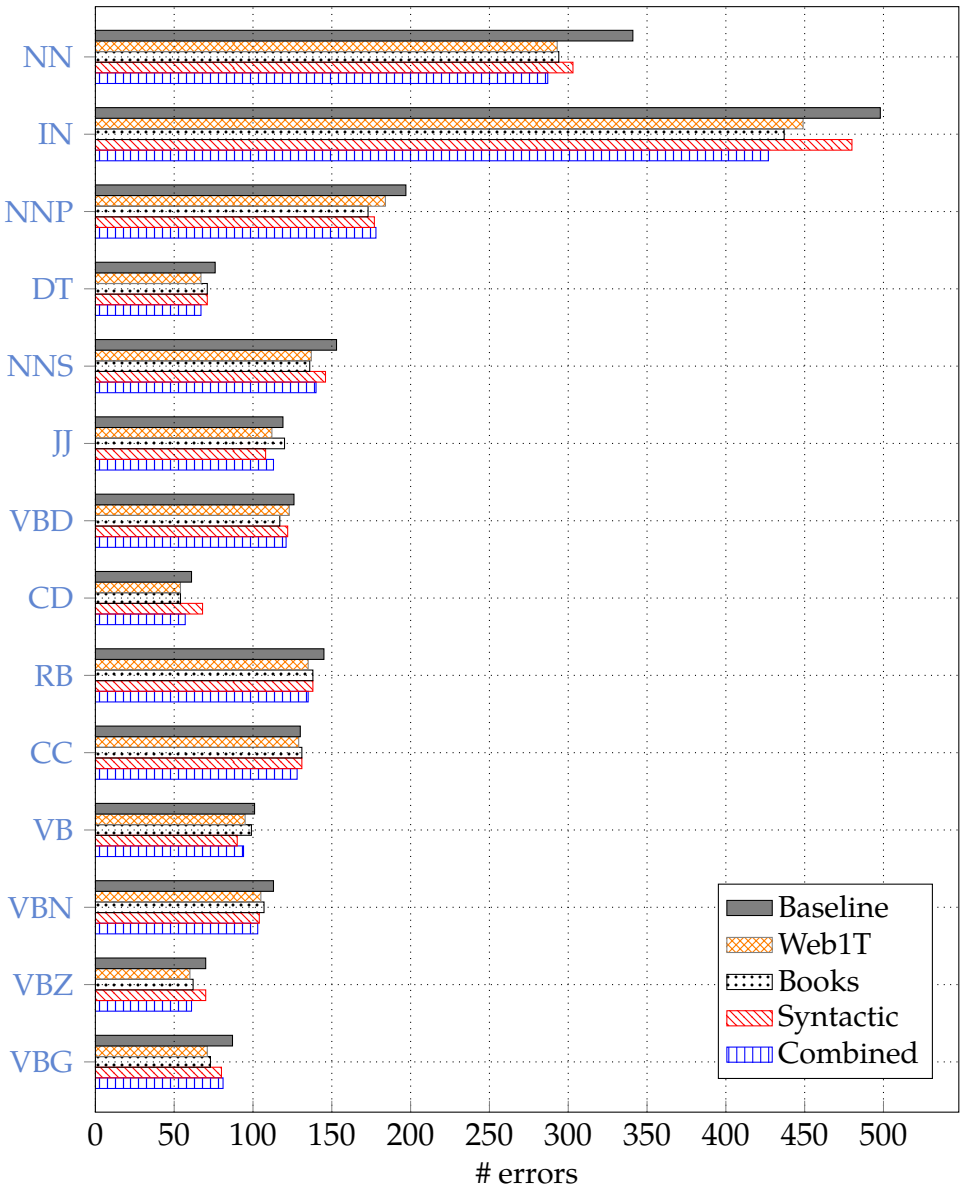


Figure 4.4: Stanford attachment errors by argument gold POS, sorted by tag frequency.

Corpus	Not present	%
Google Books	1,714,631	32.5
Web1T	1,425,347	27.0
Intersection	1,301,090	24.7

Table 4.10: Surface *n*-gram queries from the WSJ and English Web Treebank that do not receive features from Web1T and Google Books.

are challenging for syntactic *n*-grams. We found in Chapter 3 that these were two of the most substantial error classes for dependency parsers, and are the types of attachments that previous work has tried to address using new features. The Goldberg and Orwant (2013) parser did not use NP enriched training data or additional features targeted at NPs, so it is unsurprising that the syntactic *n*-gram features are less effective on NPs.

Google Books Ngrams is very evenly matched against Web1T as a source for surface *n*-grams on Stanford dependencies. We can see in Figure 4.4 that Web1T *n*-grams are slightly better on verbal POS, while Books Ngrams are ahead on nouns and prepositions. Otherwise, there are no substantial differences between the two; given that Web1T *n*-grams slightly outperformed Google Books on LTH dependencies, the differences may simply be subtle, rather than systemic variations.

4.7.1 Corpora

Web1T contains approximately double the total number of *n*-grams as Google Books, and Table 4.10 shows that 27% and 32.5% of the *n*-gram queries from the WSJ sections 02-23 (including the development and test sets) and the entire English Web Treebank do not receive features from Web1T and Google Books respectively. The intersection of these queries is 24.7% of the total, showing that the two corpora have small but substantial differences in word distributions. However, the similar performance of surface *n*-gram features from these sources suggests that the extra size of Web1T is mainly noise in this task.

We had expected our syntactic *n*-gram features to perform better than they did since they address many of the shortcomings of using surface *n*-grams. Syntactic features are sensitive to the quality of the parser used to produce them, but in this case the parser is difficult to assess as the source corpus is enormous and extracted using OCR from scanned books. Even if the parser is state of the art, it is being used to parse diverse texts spanning many genres across a wide time period, compounded by potential scanning and digitization errors. Additionally, a post-hoc analysis of the types of errors present in the corpus is impossible due to the exclusion of the full parse trees, though Goldberg and Orwant (2013) note that processing this data would almost certainly be computationally prohibitive without massive computing resources. Despite this, our work has shown that counts from this corpus provide useful features for parsing, though the magnitude of accuracy improvements is limited. Furthermore, these features stack with surface *n*-gram features, providing more substantial overall performance improvements.

4.8 Constraint-based Evaluation

Table 4.11 gives the constraint-based evaluation using our best performing Stanford dependency model. The table is comparable to Table 3.11 in Section 3.9.

We can see that on newswire text, the combined model exhibits much the same cascading behaviour as the baseline, despite outperforming it by 0.9% UAS overall. The most pronounced areas of improvement are PPs, clause attachments, NPs, and modifiers. As described in Section 4.1.2, PPs and NPs have been extensively targeted in work using web-scale *n*-gram corpora, and our results suggest that these error classes benefit the most from using *n*-gram features. The combined model regresses slightly compared to the baseline on root and coordination attachments, and improves slightly on all other attachment types.

Error class	eff	Δeff	eff %	disp	UAS	LAS	ΔUAS	Δc	Δu
WSJ section 22									
Baseline	-	-	-	-	91.3	87.5	-	-	-
NP attachment	338	-50	5.0	4.7	94.5	90.9	2.4	1.2	1.2
NP internal	199	-44	3.0	3.5	93.1	89.6	1.0	0.7	0.3
Modifier attachment	341	-29	8.0	4.0	93.9	90.7	1.8	1.2	0.6
PP attachment	359	-47	12.1	4.5	93.7	89.8	1.6	1.3	0.3
Coordination attachment	310	-6	19.0	6.5	93.7	90.1	1.7	1.1	0.6
Clause attachment	237	-29	17.4	6.9	93.5	90.0	1.4	0.8	0.5
Other attachment	294	-17	8.8	6.7	93.9	91.0	1.9	1.0	0.9
Punctuation attachment	658	-15	17.7	8.4	94.2	89.8	2.1	0.3	1.9
Root attachment	81	-4	6.1	10.2	92.9	88.8	0.8	0.3	0.6
All attachments	2817	-241	8.8	6.1	100.0	100.0	7.9	7.9	0.0
EWT development sections									
Baseline	-	-	-	-	83.1	77.4	-	-	-
NP attachment	2235	-378	8.0	4.4	88.3	82.8	3.4	2.0	1.4
NP internal	1383	-360	7.2	2.9	86.6	81.1	1.8	1.2	0.6
Modifier attachment	2340	-399	15.8	4.3	87.8	83.0	3.0	2.1	0.9
PP attachment	1915	-219	17.6	4.3	87.0	81.1	2.2	1.7	0.5
Coordination attachment	2323	-220	28.0	6.3	87.6	82.1	2.8	2.0	0.7
Clause attachment	1733	-103	26.7	7.4	87.3	82.0	2.5	1.5	1.0
Other attachment	3815	-233	21.2	5.8	89.8	85.3	5.0	3.2	1.8
Punctuation attachment	4989	-182	34.6	6.8	88.9	81.7	4.0	0.1	3.9
Root attachment	1551	-120	18.7	6.7	87.9	81.2	3.0	1.4	1.6
All attachments	22284	-2214	17.4	5.7	100.0	100.0	15.1	15.1	0.0

Table 4.11: The coverage, effective constraints and percentage, error displacement, UAS, LAS, ΔUAS over the baseline, and the constrained and cascaded Δ for the combined Stanford model over WSJ 22 and the concatenated EWT development sections.

We can also see that on newswire, the new features have little impact on cascading errors. Δu for all error classes using the combined model are within 0.1% of the baseline model. However, the largest Δc improvement made by the combined model is 0.3% for clause attachments, so there are relatively few new arcs for cascading improvement to draw from.

On web text, the combined model outperforms the baseline more substantially, at 84.8% UAS compared to 83.1%. This carries over into a wider impact on each of our error classes, with every class seeing a substantial reduction in effective constraint percentage. Particularly notable is the impact on cascading errors: the combined model substantially reduces Δu for NP attachments and punctuation attachments, from 1.8% Δu to 1.4% for the former and 4.2% to 3.9% for the latter. All other error classes again see small improvements across the board, though unlike newswire text, no error class performs worse with the combined model than the baseline.

The constraint evaluation reveals that the new features have been particularly helpful for some error classes, but have not greatly reshaped the way in which MSTParser handles each class.

4.9 Summary

Influenced by the findings in the previous chapter, we developed features for dependency parsing using subtree counts from 345 billion parsed words of scanned books. We extended existing work on surface *n*-grams from first to second-order, and compared the utility of web text and scanned books as surface *n*-gram sources. We evaluated our work on two parsing models, two dependency schemes, and across multiple domains.

Our individual feature sets all perform similarly on MSTParser, providing significant improvements in parsing accuracy of about 0.5% on newswire and up to 1.0% averaged across the web treebank domains. They are also complementary, with our best system combining surface and syntactic *n*-gram features to achieve up to 1.3% UAS

improvements on newswire and 1.6% on web text. These improvements are consistent across both LTH and Stanford dependencies.

Our constraint-based evaluation sheds further light on which error classes to which the new features are best suited. However, it also shows that the relative importance of each error class has not been changed by the features. Both surface and syntactic *n*-gram features have substantially reduced the error rate for PP, NP, and clause attachments, and assist with all error classes to some extent. The features also do not have a large impact on the constrained and cascading impact in each error class.

In the next chapter, we turn to CCG parsing, and investigate the necessity for CCG dependencies as constraints in *n*-best parsing. This will lead into the implementation of dependency-level constraints in CCG parsing.

5 Dependency Hashing for CCG

In this chapter, we move from dependency parsing to Combinatory Categorial Grammar (CCG) parsing, where a constituency-based parsing process generates dependency-based output. We identify how n-best parsing algorithms which do not account for the dependency generation process produce semantically redundant CCG parses. We develop a technique called *dependency hashing* which efficiently addresses this issue, and demonstrate its effectiveness for generating semantically distinct parses useful for downstream reranking.

We begin this chapter by surveying CCG parsing, focusing on the C&C CCG parser (Clark and Curran, 2007b) and the Brennan (2008) n-best parsing extension to C&C using the algorithms of Huang and Chiang (2005). These algorithms generate n-best parses under the assumption of *derivational difference*: the top n parses being sought are to differ from each other with respect to their structure. However, the ability of combinatory rules to license derivationally distinct by semantically equivalent derivations means that this assumption does not apply for CCG. Non-normal forms, different choices of punctuation attachment points, and order of combinator applications have no impact on the underlying dependencies generated by the derivation. We demonstrate how the Huang and Chiang (2005) algorithms cannot be directly implemented for CCG as they produce highly redundant parses which have identical dependencies, reducing their effectiveness for downstream tasks such as reranking.

A straightforward method of addressing identical dependencies is to filter out n-best parses which share the same dependencies as a previously returned parse. This

naïve method is highly inefficient, and cannot be effectively implemented in CCG parsing as dependencies may be formed in different locations in two different parses.

We develop *dependency hashing*, a technique to ensure that n-best CCG parses have unique dependencies without the cost of exhaustive comparison. Dependency hashing completely eliminates redundant parses, and allows for the efficient production of high-quality n-best parses in CCG. We present modified versions of Huang and Chiang (2005)'s algorithms implementing our dependency hashing technique for n-best CCG parsing. These algorithms are applicable to any other grammar formalism exhibiting such derivational redundancy. We demonstrate the effectiveness of dependency hashing in efficiently generating diverse n-best lists, and show how they improve a CCG reranker.

5.1 Parsing with CCG

There has been substantial research on CCG parsing. Clark et al. (2002) developed an early CCG parser with a probability model estimated over dependency structures. The two-phase design of this system incorporated a *supertagger* for assigning lexical categories, and then a *parser* using the CKY algorithm to combine the provided categories into a derivation and produce dependencies. This design would strongly influence later CCG parsers, most significantly the Clark and Curran (C&C) parser.

Hockenmaier and Steedman (2002) presented a generative model of CCG derivations based on the highly influential Collins (1999) constituency parser. CCG trees are generated top-down and head-first, with the constituents in each phrase then conditioned on the head. Additional features based on the tree structure, and the word-word dependencies projected by the constituent structure are included in the model, which is pruned using beam search. The parser outperformed Clark et al. (2002), despite not modelling every dependency in the logical form. Hockenmaier (2003b) extended the generative model to capture the CCG predicate-argument dependencies in CCGbank.

Clark and Curran (2003, 2007b) followed the Clark et al. (2002) system design, but applied discriminative log-linear models to both tagging and parsing. This allowed for rich, arbitrary, and overlapping features to be seamlessly captured and integrated together in the tagging and parsing models. *Adaptive supertagging*, whereby the supertagger first provides a very limited set of categories, and progressively provides more if the parser cannot combine them together leads to a highly efficient and accurate system. The resulting C&C parser records a best labeled F-score of 87.7% over CCGbank section 23 on gold POS tags, and 85.5% on automatic POS tags, compared to 84.4% and 83.3% respectively for the Hockenmaier (2003a) parser. C&C is the focus of the work in this chapter, and it will be discussed in detail in Section 5.2.

Fowler and Penn (2010) observe that CCG rule instantiations from a corpus such as CCGbank can be interpreted directly as phrase-structure rules, and also prove that the set of such rules extracted from CCGbank is strongly context-free. They test the implication that standard Penn Treebank-style parsers could be directly trained on CCGbank by applying the Petrov and Klein (2007) split-merge Berkeley parser to the corpus. By converting the resulting CCG derivations to dependencies, they demonstrate a small performance improvement over the C&C parser,[1] but with a system that is not specifically designed for CCG. However, their parser is also several hundred times slower than C&C.

Zhang and Clark (2011a) implement a wide-coverage shift-reduce transition-based parser for CCG. The transition actions for the parser are extracted from the rule instantiations in CCGbank in a similar manner to Fowler and Penn (2010). Categories are assigned to words using the C&C supertagger. Similar to ZPar, used in Chapter 3, this system employs the averaged perceptron (Collins, 2002) to predict the best transition actions, with the top 16 actions kept in a beam to allow limited backtracking from poor decisions. Following a conversion of its output to CCG dependencies, the system achieves 85.5% labeled F-score on automatic POS tags over CCGbank section 23. This is

[1] A direct comparison is difficult due to coverage differences between the systems.

very competitive with the C&C parser, despite the enormous ambiguity in transition actions for shift-reduce CCG parsing and the use of a small beam.

Merity and Curran (2011) describe another shift-reduce CCG parser, built as an extension upon the C&C parser. The key difference is the use of a graph-structured stack to explore all possible transition sequences in polynomial time, rather than using heuristic linear-time greedy or beam search. This system is statistically indistinguishable from the baseline C&C parser, though slightly slower due to the highly engineered nature of the baseline parser. However, the graph-structured stack allows for unlikely derivations to be pruned at the frontiers of the parsing process, substantially improving parsing speed at a very small cost in accuracy.

Auli and Lopez (2011b) extend the generative CCG parsing model of Hockenmaier and Steedman (2002) by replacing beam search with the A* search algorithm and heuristic of Klein and Manning (2003a). They showed that A* combined with the adaptive supertagging of Clark and Curran (2007b) could reduce the amount of work required to build a parse compared to the exhaustive CKY algorithm. Lewis and Steedman (2014a) also use A* for a very simple CCG parsing model factored on category assignments to the words in a sentence. The parsing process is a search for the most probably category sequence licensing a valid CCG derivation, admitting an efficient heuristic for A*. The system is four times faster than C&C, whilst being slightly less accurate on newswire and slightly more accurate on out-of-domain text. However, parsing times increase linearly with the length of sentences, and the parser itself does not model or produce the underlying CCG derivations or dependencies.

Xu et al. (2014) build on the work of Zhang and Clark (2011a) in developing a dependency-based model for beam-search shift-reduce CCG parsing. An important feature of the system is the use of a dependency oracle during training (similar to Goldberg and Nivre (2012)), which allows dependencies to be modeled directly. They observe significant performance improvements over the C&C parser and the Zhang and Clark (2011a) parser, with a labeled F-score of 86.1% on CCGbank section 23 with

automatic POS tags and a beam size of 128. Despite the use of a linear-time algorithm, both this parser and Zhang and Clark (2011a) are slower than C&C, as the latter is highly engineered with a focus on speed.

In this and the following chapter, we investigate dependency-level constraints for a variety of tasks in CCG parsing. This requires a parser which generates dependencies as it builds the derivation, rather than first producing a full derivation and then extracting the dependencies from it. In fact, most other CCG parsers in the recent literature use the `generate` tool included with C&C to convert derivations to CCG dependencies. Thus, we have used the C&C parser for the work in this chapter, though our investigations are broadly applicable to CKY-based CCG parsing in general.

5.2 The C&C Parser

The C&C parser (Clark and Curran, 2003, 2007b) is a fast, wide-coverage CCG parser. It is highly efficient, with a design strongly influenced by CCGbank, the corpus on which it is trained for English.

5.2.1 Supertagging

The C&C parser processes sentences using a two-phase process: first, CCG categories are assigned to each word in the sentence, and then the categories are combined together to form a spanning analysis over the sentence. In C&C, a *supertagger* performs the first step, while the *parser* takes the categories produced by the supertagger to form an analysis using the CKY algorithm (described further in Section 5.2.2).

The supertagging process has been described as *almost parsing* (Bangalore and Joshi, 1999) since the lexical categories assigned by the supertagger encode detailed syntactic information. As described in Section 2.3.1, categories describe the form and arity of their arguments, constraining the combinations that the parser can employ. With perfect categories, the parser only has to determine how to combine them to form a

correct analysis. The C&C supertagger is a maximum-entropy sequence tagger that can process sentences in time linear in the number of words and quadratic in the category set size. The system uses a restricted set of 425 categories determined by a frequency cut-off of 10 in CCGbank sections 02-21. The category set has wide coverage, but still allows the supertagger to run efficiently, dramatically cutting down on the 1,286 total categories in CCGbank (Clark and Curran, 2004b).

The C&C supertagger achieves around 92% accuracy on CCGbank section 00 (Curran et al., 2006). Unfortunately, sentence accuracy (where all words in a sentence are correctly tagged) is only 36.8%, which is too low for parsing — it is likely that the parser will be unable to find any analysis for the sentence given so many incorrect categories. To address this problem, C&C utilises *multi-tagging*, returning multiple categories within some factor β of the highest probability category for each word. The multi-tagger increases per-word accuracy to 98.5% and sentence accuracy to 78.4% using all categories within 1% of the most probable category. On average this assigns 2.9 categories per word (Clark and Curran, 2004b).

The C&C supertagger also uses a *tag dictionary*, as described by Ratnaparkhi (1996), and accepts a cutoff k to control this dictionary. Words seen more than k times in training may only be assigned categories seen with that word more than 5 times, and the category frequency must also be no less than 1/500th of the most frequent category for that word. Words seen fewer than k times may only be assigned categories seen with the POS tag of the word in training, subject to the cutoff and ratio constraint (Clark and Curran, 2004b). The tag dictionary eliminates infrequent categories and improves supertagger performance, but at the cost of removing unseen or infrequently seen categories from consideration.

During parsing, the supertagger is initially run with a larger β value, which has the effect of only allowing categories with a probability close to that of the most likely to be returned. This assigns a highly restricted set of categories to each word, and the parser attempts to find an analysis using this small set of categories. If an analysis

cannot be found, the supertagger is run again with a smaller, less restrictive β value to return more categories per word. The parser then tries to find an analysis again using this larger category set. This process continues for a fixed number of iterations until a spanning analysis is found; if no analysis if found, or if the total number of tree nodes exceeds a predefined limit, the parse fails.

Clark and Curran (2007b) show that this tight integration between parser and supertagger leads to very efficient parsing with wide coverage and high accuracy. In particular, the restricted set of categories that are initially provided to the parser serve to prune the search space of highly unlikely options, allowing them only if the parser finds that it needs them. This helps the parser deal with the substantial ambiguity of words with many syntactic forms. Off-loading some of the disambiguation work to the supertagger also increases parsing speed, as the supertagger has far lower time complexity than the parser itself.

One potential issue with two-phase parsing is that if the supertagger does not provide the correct categories to the parser, then the parser cannot possibly recover the best parse. Supertagger accuracy is not perfect, and while multi-tagging improves the category recall, there are still tags which are missed. It has been an ongoing consideration as to how much of the parser's error is caused by the supertagger, and how much is caused by deficiencies in the parser model. One of the contributions of this thesis is a thorough examination of this issue.

5.2.2 Parsing CCG with the Cocke-Kasami-Younger Algorithm

The Cocke-Kasami-Younger (CKY) algorithm (Cocke and Schwartz, 1970; Kasami, 1965; Younger, 1967) is a bottom-up dynamic programming algorithm that can be used to identify whether a given sentence is parseable under a provided context-free grammar. CKY works by constructing all possible trees licensed by the grammar given a sentence. A triangular data structure called a *chart* is used to cache equivalent subtree nodes and their scores, allowing them to be reused as a sentence is analysed. Steedman (2000)

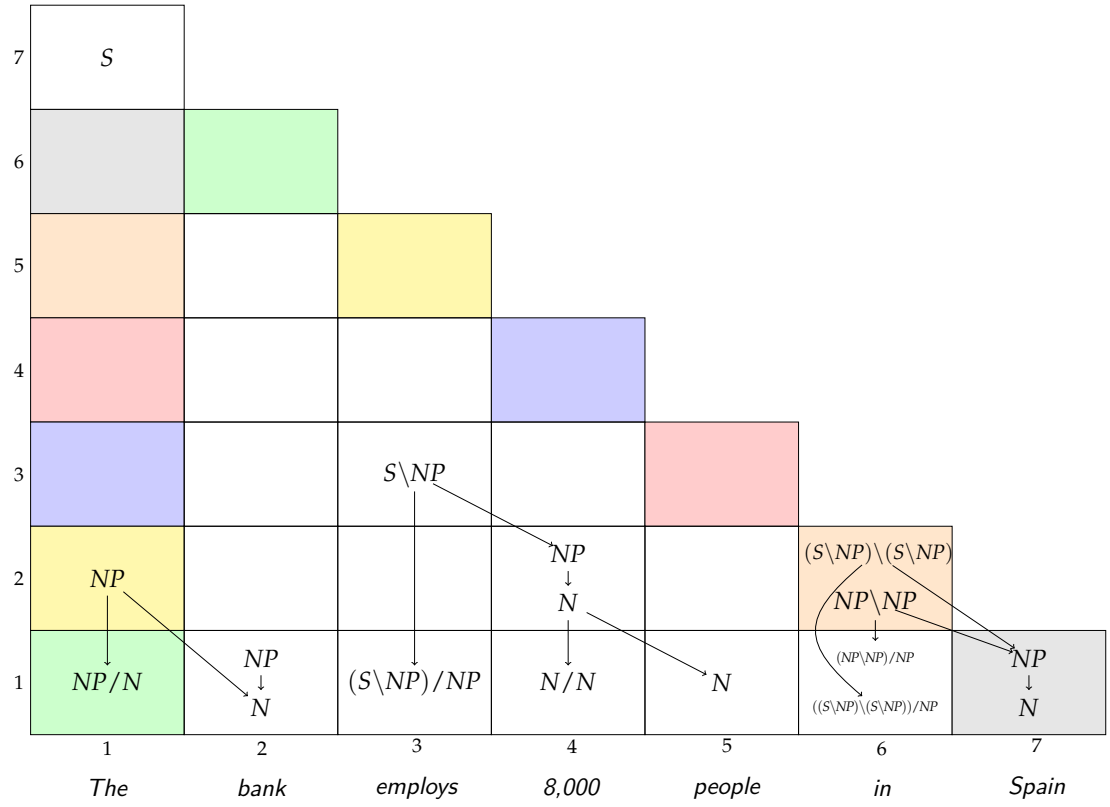


Figure 5.1: A partially filled CCG chart.

observes that the binary branching nature of CCG lends itself well to CKY parsing, and Section 5.1 describes several CCG parsers which use this algorithm, including the C&C parser.

In the chart, the horizontal indices represent the *position* (*pos*), and indicate the start index of a constituent. The vertical indices represent the *span* and indicate the size of the yield of a constituent. Each block in the chart is called a *cell*. A cell with pos i and span j contains subtree nodes covering the tokens at indices $[i, i + j)$ in the sentence. For example, the cell containing S in Figure 5.1 spans over the entire sentence as it begins at position 1 with span 7. Conversely, the cells with span 1 cover only each individual word.

A key optimisation is collapsing equivalent categories in a cell into *equivalence classes* to produce a *packed chart* (Miyao and Tsujii, 2002). Each equivalence class contains all categories with the same position, yield, active variables, and unfilled dependencies;

these categories which will behave identically to the rest of the derivation, though each individual member of the equivalence class may yield a different derivation and different dependencies. This reduces the number of possible combinations which the parser must consider.

Each equivalent subtree node in each cell is represented by an equivalence class. Every result category within an equivalence class is the product of different derivations and combinatory rules, so each records an independent set of the following data:

- *backpointers* to the child equivalence classes that were combined;
- a set of *active variables*, co-indexed with words in the sentence that are are available (or desired) as heads or fillers for dependencies. They may be *assigned* or *unassigned*.
- a set of *unfilled dependencies*, specifying the CCG dependencies to be created when unassigned active variables are unified and given a value during category combination;
- a set of *filled dependencies*, listing dependencies generated by the combinatory or unary rule creating the category.

The full set of dependencies generated by a CCG derivation in the parser is found by enumerating the maximum scoring tree using the backpointers on each result category, collecting the filled dependencies at each category in the derivation.

When a cell (i, j) is built, the contents of all pairs of non-overlapping cells that together span $[i, i + j)$ are considered. The use of dynamic programming ensures that each cell is only built once and memoised for further use; this also ensures that considering pairs of non-overlapping cells is sufficient for enumerating all possible analyses. In Figure 5.1, the pairs of colour-coded cells are those which are considered when building the S cell — that is, these combinations of cells begin at position 0 and span the entire sentence.

The chart parsing process in the C&C parser proceeds bottom-up from spans of length 1 up to the sentence length. The categories from the supertagger are added to the bottom row of cells in the chart (i.e. the cells with span 1). Then, for each increasing span length j and position i, all possible pairs of equivalence classes (a_x, a_y) from pairs of cells (x, y) that can cover the span $[i, i + j)$ are considered by the parser. Combinable pairs have their result category added to the appropriate equivalence class in (i, j). The active variables from each child category are unified according to the combinator used, any unfilled dependencies from the children that have an assigned variable are marked as filled, and the new list of variables and dependencies are passed to the new result category.

Once all pairs of possible children are considered, unary rules are applied to the resulting categories in the chart cell, with their results added to the appropriate equivalence classes in the cell as well. Unary rules have a single child from the same cell as the result rather than two children from two different cells. The C&C parser implements a subset of the unary rules present in CCGbank (described in Section 2.4.3), as well as a limited set of CCG type-raising rules. The lack of full coverage contributes to *grammar errors* that will be further discussed in the error analysis in Chapter 7.

A number of techniques are utilised in C&C to manage the proliferation of subtree nodes and increase parsing speed. First, the *seen rules* constraint allows the parser to only combine categories if they were seen to combine in CCGbank sections 02-21. Second, the Eisner (1996b) *normal-form rules* (described in Section 2.3.4) are used to contain the productivity of the composition and type-raising combinators, limiting the impact of CCG's spurious ambiguity. Finally, the *maxsupercats* value gives the maximum possible number of subtree nodes which may be added to the chart before the parse fails.

When the parser has finished filling the chart, all valid derivations over the sentence will be found in the topmost cell. The chart is then *decoded* according to a parsing model;

Algorithm 5.1 CKY CCG *Parsing*

Require: A sentence of length n

Ensure: All possible spanning analyses of the sentence

```
for all i from 0 to n do
    assign one or more categories from supertagger to cell (i, 1)
▷ iterate over spans
for all j from 1 to n do
    ▷ iterate over pos
    for all i from 0 to n − j do
        ▷ iterate over split points
        for all pairs of cells x, y that span (i, i + j) do
            for all equivalence classes e_x, e_y in cells x, y do
                if e_x and e_y can be combined with a rule r to produce category c then
                    unify variables from e_x and e_y to c
                    add filled dependencies from e_x and e_y to c
                    add backpointers to e_x and e_y to c
                    add c to the appropriate equivalence class in cell (i, j)
        for all equivalence classes e_x in cell (i, j) do
            if a unary rule r can be applied to e_x to produce category c then
                unify variables from e_x to c
                add filled dependencies from e_x to c
                add backpointers to e_x to c
                add c to the appropriate equivalence class in cell (i, j)
return the root cell (0, n)
```

each equivalence class is scored, with the optimal option at each node in the derivation combined together to form the best-scoring derivation using dynamic programming.

Algorithm 5.1 describes the CKY algorithm in the C&C parser.

5.2.3 Decoding

Once the chart is filled, it is then *decoded* according to a statistical model to find the highest scoring derivation. Clark and Curran (2007b) describe a number of discriminative maximum entropy parsing models for the C&C parser, including a model based on CCG dependency structures, a model based on the normal-form derivations in CCGbank, and a hybrid model combining both. The dependency model involves summing the

Algorithm 5.2 $decode_{\text{nf}}$

Require: A filled chart *chart*
Ensure: The most probable spanning analysis

```
max_e ← ∅
max_score ← 0
for all categories c in the root cell of chart do
    score, equiv ← best-equiv (c)
    if score > max_score then
        max_score ← score
        max_e ← equiv
return max
```

probabilities of all possible derivations that yield a particular dependency structure, including the non-standard derivations eliminated by normal-form constraints. The normal-form model scores features only on from the derivations.

Algorithm 5.3 *best-equiv*

Require: An equivalence class *e*
Ensure: The best member category of the equivalence class

```
if e.max exists then
    ▷ memoising for dynamic programming
    return e.score, e.max
max ← ∅
max_score ← −∞
for all categories c in e do
    score ← best-score (c)
    if score > max_score then
        max_score ← score
        max ← c
e.max ← max
e.score ← max_score
return max_score, max
```

Algorithms 5.2 to 5.4 describe the Viterbi decoding process for the C&C normal-form model. The dependency model and hybrid models are decoded through a different algorithm which aims to maximise the expected recall of dependencies in the final analysis. As we exclusively use the normal-form model and decoder in this work, we

Algorithm 5.4 ***best-score***

Require: A category c in the chart
Ensure: The best score for the subtree rooted at c

```
1  ▷ use features on the current local subtree
2  score ← SCORE(c)
3  if c has a left child c.left then
4      ▷ combined with scores on children
5      s, _ ← best-equiv (c.left)
6      score ← score + s
7      if c has a right child c.right then
8          s, _ ← best-equiv (c.right)
9          score ← score + s
10 return score
```

omit a detailed description of the dependency decoder, though our contributions are naturally applicable to it as well.

5.2.4 Parser Performance and Output

Table 5.1 summarises the normal-form and hybrid parsing model performance of the C&C parser. C&C accepts POS-tagged text as input; these tags are fixed and remain unchanged throughout during the parsing pipeline. The POS tags are important features for the supertagger; parsing accuracy using gold-standard POS tags is typically 2% higher than using automatically assigned POS tags (Clark and Curran, 2004b).

The hybrid model outperforms the normal-form model by 0.3-0.4% F-score across all experiments. However, the normal-form model is easier to experiment with, as it is faster to train and modify. All experiments in this thesis use the normal-form model with the Viterbi decoder described previously.

The C&C parser provides output in several different formats. Given a sentence, it can produce the normal-form CCG derivation, the CCG dependencies generated by the derivation, and a sequence-tag style output of the lexical categories chosen by the parser for each word in the sentence.

	Gold POS		Automatic POS	
Corpus	LF	coverage	LF	coverage
Normal-form				
CCGbank 00	86.84	99.16	84.91	99.06
CCGbank 23	87.36	99.58	85.07	99.58
Hybrid				
CCGbank 00	87.20	99.16	85.20	99.06
CCGbank 23	87.68	99.58	85.50	99.58

Table 5.1: The labeled dependency recovery F-score and coverage of the normal-form and hybrid C&C models under gold and automatic POS tags.

Clark and Curran (2007a) describe a formalism-independent comparison of C&C against the Briscoe et al. (2006) RASP parser on DepBank. In order to perform this evaluation, Clark and Curran develop a many-to-many mapping from CCG dependencies to the GR format used by RASP. GRs provide a more formalism-neutral output, and abstract away from the raw CCG dependencies and most importantly from the CCG categories in each dependency. For instance, the CCG dependency with a subject NP to the left of an intransitive, transitive, or ditransitive verb would differ from each other. The object relationship with an NP to the right of a transitive or ditransitive verb would also be different depending on the verb. GRs allow generalisations such as *subject* and *object* relationships to be extracted from all of these instances.

Despite both parsers producing head-argument dependencies, producing the mapping was a complex and time-consuming process, requiring the annotation of each CCG category used in the parser with a template describing the GR (or multiple GRs) which the head and slot fillers would be converted into. However, C&C compared very favourably against RASP in the evaluation, outperforming the unlexicalised parser by over 5% overall despite an upper bound of 84.8% accuracy imposed by the conver-

sion process. As the mapping to GRs is integrated into the system, C&C can natively produce GRs as well.

Dependency and GR creation in the C&C parser is controlled by the *markedup* file, which describes the co-indexation annotations, slots, and dependency to GR mapping described in the previous section. Changes to the markedup file will change the dependencies that the parser can form, as well as the unification mechanism through which categories are combined together. Rimell and Clark (2008) describe changing the markedup file to allow C&C to produce Stanford dependencies, though substantial additional post-processing was required to align the output with the scheme.

5.2.5 Extensions to C&C

Each time that C&C is unable to find a spanning analysis with the categories supplied by the supertagger, it asks for supertagger for more categories, and tries the process again. However, as the new categories are a superset of the previous, the parser is guaranteed to construct at least the same derivations as the previous iteration. Instead of discarding the chart and rebuilding it entirely, Djordjevic (2006) introduced *chart repair*, where only the cells affected by the new categories are rebuilt as necessary.

Chart repair allows the derivations in the chart to be reused across the parser's iterations. Following a reattempt, the chart is retained, and any new categories from the supertagger are added to the bottom row as usual. When the parser combines cells during CKY, it can skip any point where the two cells involved have not had any new categories added since the previous iteration. Otherwise, it performs the combination, and marks the result cell as having new categories. No other changes to the parser or decoder are required, as a reattempt only occurs when the parser cannot find a spanning analysis. At least one new equivalence class must be created somewhere in the chart for a spanning analysis to be found; adding a new category to an existing equivalence class cannot possibly create a spanning analysis as the class was available

during the previous attempt. Thus, the scores and optimal ancestor pointers calculated during decoding can be cached and reused in the reattempt without issue.

On average, chart repair updates one third of the cells in the chart, and avoids the redundancy of having to rebuild many of the same derivations. The technique contributes to an 11% speed increase with no reduction of parsing accuracy.

Djordjevic (2006) also implements *span-level constraints* in the C&C parser. These constraints indicate cells in the chart which must be used at some point in the derivation. For example, a constraint may exist at span 3 and position 4, indicating that the cell at that point in the chart must be used as part of a combinatory rule application in an spanning analysis. These constraints are analogous to requiring that certain *constituents* are present in the final CCG derivation.

The most successful constraint they used was a punctuation constraint, enforcing that sentence final punctuation had to attach to the root. While this constraint had no significant impact on parsing accuracy, it substantially increased parsing speed by preventing the productive punctuation rules from overgenerating. More general constraints, such as NP boundaries derived from the output of a chunker, were less effective, and substantially decreased parsing accuracy due to misalignments between CCG constituents and those produced by the chunker.

Kummerfeld et al. (2010) describe a novel self-training technique to dramatically improve parsing speed. In the baseline C&C parser, the supertagger provides a set of possible categories for each word in the sentence to the parser, which attempts to find a spanning analysis using those categories. Kummerfeld et al. make the observation that by retraining the supertagger on the categories that the parser eventually uses (not necessarily the *correct gold-standard* categories), the supertagger can be "adapted" to the parser. This effectively trains the supertagger to provide the categories that the parser would use anyway, leading to speed increases of up to 85% without no impact on accuracy. More importantly, the extra speed could be traded off for greater accuracy by relaxing the β values used by the supertagger.

Auli and Lopez (2011a) note that adaptive supertagging both prunes the search space for the C&C parser of many bad alternatives, but also removes desired categories from the parser's view. To address this, they use loopy belief propagation and dual decomposition to integrate the CCG supertagging phase into the parser itself. The intuition is that this will give the parser more information to work with, as well as prevent the supertagger from pruning good parses prematurely. Their best system achieves an F-score of 86.7% on automatic POS tags over a reduced subset of CCGbank 23, a 1% increase over the baseline C&C parser. On a similar theme, Lewis and Steedman (2014b) describe a variety of semi-supervised approaches to supertagging, using vector-space embeddings for features with neural network and CRF models. By applying these models in place of the C&C supertagger, they achieve 86.1% F-score on automatic POS tags over the full section 23.

Auli and Lopez (2011c) replaces the usual log-linear loss function used for training the C&C parser with a variety of other loss functions. Most notably, they implement a softmax-margin that optimises the training process using the F-score as an objective. They report an F-score of 87.2% over a reduced subset of CCGbank 23 on automatic POS tags, the highest reported results for CCG parsing. However, the improved accuracy comes with a very substantial speed and complexity penalty during training and testing,[2] which makes the modified parser unsuitable for large scale tasks.

5.3 Experimental Setup

We use the normal-form model and Viterbi decoding for the C&C parser described in Section 5.2.3, as well as the POS tagger distributed with the C&C parser. We trained the POS tagger, supertagger, and parser over CCGbank sections 02-21 using the scripts and methodology described in Clark and Curran (2007b), and used those models for all experiments.

[2]Michael Auli, p.c.

Level	β	k
1	0.075	20
2	0.03	20
3	0.01	20
4	0.005	20
5	0.001	150

Parameter	Value
`--parser-seen_rules`	true
`--parser-eisner_nf`	true
`--parser-maxsupercats`	1000000
`--super-forward_beam_ratio`	0.1

Table 5.2: The default β, k, and other parameters used in the C&C parser.

We ran the parser following the configuration used by Clark and Curran in their evaluation process. Table 5.2 lists the default β and k values for the C&C supertagger used, as well as the value of all other pertinent parameters passed to the parser.

5.4 n-best Parsing and Reranking

The number of possible parses typically grows exponentially in the length of the sentence, making exhaustive search in parsers intractable. Parsers instead use dynamic programming or beam-search heuristics, extracting features from a limited context around each node in the tree, and either memoising scores for later reuse or discarding low-scoring states as unlikely.

Expanding the context used by parsers increases the parser search space and decreases efficiency. While excluding non-local and global features helps make parsing tractable, they omit potentially useful long-range information from consideration. To counter this, a parser can be modified to produce the top n parses of a sentence, ordered by the probability under its model, and an external *reranker* used to reorder those parses. As the reranker only sees a limited subset of the search space considered by the parser, it can tractably include arbitrary rich features over the entire parse.

Given an n-best list of parses, the *oracle score* is the evaluation score that the parser would achieve if the best possible parse in the list was selected (Ratnaparkhi, 1997).

Higher oracle scores provide more potential for reranking, and so the success of the technique is predicated on high-quality *n*-best parse.

The major difficulty of *n*-best parsing is to retain the top *n* analyses whilst not making the parsing task intractable. Collins (2000) extends his parser to produce *n*-best output by disabling dynamic programming, and using beam search to prune analyses instead. When parsing is completed, the analyses remaining in the beam are returned. While the resulting parser produces an average of 27 parses per sentence, the method is inefficient and inexact. Beam search is an approximation which may discard high-quality parses if they do not score highly enough under the parser model.

The reranker built on the Collins (2000) *n*-best parser established conventions for developing such systems. *n*-best parses of the PTB sections 02-21 are generated by jackknifing: iteratively holding out two sections, training the parser on the remaining data, and using the model to produce *n*-best parses of the held-out sections. This data is then used to train the reranker. The features used in the system include lexical heads and the distances between them, context-free rules in the tree, *n*-grams and their ancestors, and parent-grandparent relationships. The reranker improves the accuracy of the Collins parser from 88.20% to 89.75%.

Charniak and Johnson (2005) use a *coarse-to-fine* approach for *n*-best parsing that retains dynamic programming. Initially, a coarse-grained grammar is used to find a crude parse for a sentence, then this parse is iteratively refined whilst keeping the top *n* most probable partial analyses. The initial coarse parse dramatically reduces the number of analyses that must be considered. This model reliably returns at least 50 parses per sentence, and has an oracle F-score of 96.8% — substantially higher than the 94.9% scored by the Collins parser on the PARSEVAL metric (Huang and Chiang, 2005). The accompanying reranker is trained using a similar setup to Collins, and includes additional linguistic features based on subject-verb agreement, *n*-gram local trees, and right-branching factors. In 50-best mode the parser has an oracle F-score of 96.8%, and the reranker produces a final F-score of 91.0% (compared to an 89.7% baseline).

The n-best algorithms of Huang and Chiang (2005) are the standard for generating n-best lists, and we will describe them in further detail in Section 5.4.1. These algorithms have seen uses outside of parsing; Huang et al. (2006) develop a translation reranking model using them, but faced the issue of different derivations yielding the same translated string. This was overcome by storing a hashtable of strings at each node in the tree, and rejecting any derivations that yielded a previously seen string.

Reranking has not found universal success across parsers. Johnson and Ural (2010) shows that using the reranker of Charniak and Johnson (2005) on the Berkeley split-merge constituency parser (Petrov and Klein, 2007) results in negligible accuracy improvements. In work prior to this thesis, we found that a direct implementation of the Charniak and Johnson (2005) reranker for the C&C CCG parser performed poorly. Substantial tuning and CCG-specific features were required to achieve relatively small performance improvements of 0.2% F-score (Ng et al., 2010). The question of why so much adaptation was required for such a small F-score gain inspired the work in this chapter.

5.4.1 The n-best Algorithms of Huang and Chiang (2005)

Brennan (2008) describes an implementation of the n-best parsing algorithms of Huang and Chiang (2005) for the C&C CCG parser. The algorithms were originally described in terms of a general hypergraph parsing framework, and require some adaptation to correctly implement them for CCG. In this section, we describe the n-best algorithms as implemented by Brennan (2008).

5.4.1.1 Algorithm 0: Naïve n-best Parsing

The 1-best CKY algorithm for CCG combines compatible categories together, collapsing equivalent results into equivalence classes in each cell.

In n-best parsing, we must now store n sets of backpointers, representing the best n ways of forming each category, rather than a single pair in 1-best parsing. We

Algorithm 5.5 $\boldsymbol{mult}_{\mathrm{n}}$

Require: Two n-best lists $a_c = a_1, a_2, ..., a_n$ and $b_c = b_1, b_2, ..., b_n$ for categories a and b, where a and b combine into a result category r
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: An n-best list r_c for r

```
1 r_c ← array
2 for all a_i in a_c do
3     for all b_i in b_c do
4         score ← SCORE(r, a_i, b_i)
5         add (r, a_i, b_i, score) to r_c
6 sort r_c descending by score
7 return the first n elements of r_c
```

Algorithm 5.6 $\boldsymbol{merge}_{\mathrm{n}}$

Require: Sorted n-best lists $a_c = a_1, ..., a_n$ and $b_c = b_1, ..., b_n$ for a result category r
Ensure: An n-best list r_c for r

```
1 r_c ← array
2 while r_c does not contain n items and a_c or b_c have items do
3     (a, x_a, y_a, score_a) ← a_1
4     (b, x_b, y_b, score_b) ← b_1
5     if score_a > score_b then
6         remove a_1 from a_c and add it to r_c
7     else
8         remove b_1 from b_c and add it to r_c
9 return r_c
```

must also store the score of each option, as the n-best list for each equivalence class is sorted in descending order according to the scores. We must maintain this list for each equivalence class, as all categories in a class are equivalent to each other. The key operation is efficiently generating the n-best list of options when we combine two categories together. As each child category has its own backpointer n-best list, the simplest way to perform this is to generate all n^2 possible pairs (taking one option from each child), sort the pairs based on their score, and keep the best n of these. This procedure is called $\boldsymbol{mult}_{\mathrm{n}}$, and is described in Algorithm 5.5.

Once we have generated a new result category and its n-best list, it will be added to the appropriate equivalence class in the chart cell. If there is already a category in

Algorithm 5.7 ***new-mult***$_{\text{n}}$

Require: Two n-best lists $a_c = a_1, a_2, ..., a_n$ and $b_c = b_1, b_2, ..., b_n$ for categories a and b, where a and b combine into a result category r
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: An n-best list r_c for r

```
1 r_c ← array
2 cand ← max-heap
3 add (r, a_1, b_1, SCORE(r, a_1, b_1)) to cand
4 while r_c does not contain n items and cand contains items do
5     (r, a_x, b_y, score) ← the first item in cand
6     add (r, a_{x+1}, b_y, SCORE(r, a_{x+1}, b_y)) to cand
7     add (r, a_x, b_{y+1}, SCORE(r, a_x, b_{y+1})) to cand
8     add (r, a_x, b_y, score) to r_c and remove it from cand
9 return r_c
```

the equivalence class, there will already be a sorted n-best list of backpointers, and so the existing and the new lists can be merged together in a procedure analogous to mergesort called ***merge***$_{\text{n}}$.

Adding ***mult***$_{\text{n}}$ and ***merge***$_{\text{n}}$ to the 1-best CKY algorithm gives us Huang and Chiang (2005)'s Algorithm 0 for n-best parsing.

5.4.1.2 Algorithm 1: Speeding up *mult*$_{\text{n}}$

An obvious efficiency improvement for Algorithm 0 is to recognise that each n-best list is sorted in descending order by score. When generating the new n-best list for a new result category r from its child categories a and b, the local score at r must also be the same for all n-best combinations, as they form an equivalent category. Thus, it is not necessary to consider all n^2 possible combinations from a_c and b_c. Consider two n-best lists $a_c = a_1, a_2, ..., a_n$ and $b_c = b_1, b_2, ..., b_n$ for a result category r. The best option must be (a_1, b_1), as these are the highest scoring options for producing a and b. Then, the next best option must be one of the two immediate *successors* to the best: either (a_2, b_1) or (a_1, b_2), and so forth.

Algorithm 5.8 ***find-best***$_{\mathrm{n}}$

Require: An equivalence class e
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: An n-best list e_c for e

```
1 e_c ← array
2 cand ← get-candidates_n(e)
3 while e_c does not contain n items and cand contains items do
4     (r, a_x, b_y, score) ← the first item in cand
5     add (r, a_{x+1}, b_y, SCORE(r, a_{x+1}, b_y)) to cand
6     add (r, a_x, b_{y+1}, SCORE(r, a_x, b_{y+1})) to cand
7     add (r, a_x, b_y, score) to e_c and remove it from cand
8 return e_c
```

In Algorithm 1 of Huang and Chiang (2005), a *candidate list* is maintained, containing the successors of each n-best option. The candidate list is sorted in descending order by score, as with the existing n-best lists. As a candidate is taken and added to the n-best list, its two successors are then added to the candidate list in descending order by score. This procedure is called ***new-mult***$_{\mathrm{n}}$, and it reduces the quadratic factor of enumerating all possible n-best candidates to linear, whilst incurring the extra logarithmic factor of maintaining a sorted candidate list.

5.4.1.3 Algorithm 2: Candidate Lists of Equivalence Classes

We now turn our attention to ***merge***$_{\mathrm{n}}$. Even though we have sped up the n-best list generation for each category, we must still repeatedly generate and merge lists for each equivalence class in each cell. However, this is wasteful, as only the best n ways of constructing *each* equivalence class in the cell will ever be required.

We can use this to improve the efficiency of candidate generation. Rather than iteratively generating a new list for each member of each equivalence class, we can generate a single n-best list for the entire equivalence class at once. To do this, we adapt ***new-mult***$_{\mathrm{n}}$ to maintain a *candidate list* over all categories in an equivalence class, iteratively adding the best option and then generating its successors. This combines ***merge***$_{\mathrm{n}}$

Algorithm 5.9 ***get-candidates***$_{\text{n}}$

Require: An equivalence class e
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: A candidate list *cand* for e

```
1 cand ← heap
2 for all categories r in e do
3     a_c ← the n-best list from r's left child
4     b_c ← the n-best list from r's right child
5     score ← SCORE(r, a_1, b_1)
6     add (r, a_1, b_1, score) to cand
7 return cand
```

Algorithm 5.10 ***lazy-best***$_{\text{n}}$

Require: A chart C with a 1-best analysis decoded in $C.root$
Ensure: An n-best list of best spanning analyses for the chart

```
1 return lazy-best-equiv_n (C.root, n)
```

and ***new-mult***$_{\text{n}}$ together, and substantially reduces the number of n-best candidates which must be considered. This procedure is described in Algorithms 5.8 and 5.9.

5.4.1.4 Algorithm 3: Lazy Computation

Algorithm 2 works well when each equivalence class is large. However, it still calculates an n-best list for every class in every cell of the chart, regardless of whether or not the class actually appears in the final n-best list of parses for the sentence. Addressing this deficiency requires lazy computation: running the 1-best parsing algorithm to completion, and then computing the n-best list for only the required equivalence classes.

Algorithms 5.10 to 5.12 describe the lazy n-best parsing approach of Algorithm 3. The important insight is that at any point in the 1-best derivation, the next best option must differ from the 1-best in one location only. The main loop of ***lazy-best-equiv***$_{\text{n}}$, which drives the process, lazily generates the next best candidate for each equivalence class in the derivation *as it is required* by calling ***lazy-next***$_{\text{n}}$. Then, the best existing candidate is removed from the candidate set and added to the n-best list,

Algorithm 5.11 ***lazy-best-equiv***$_{\mathrm{n}}$

Require: An equivalence class e
Require: A limit l
Ensure: An n-best list e_c for e

```
1  e_c ← array
2  if e.cand does not exist then
3      e.cand ← get-candidates_n(e)
4  while e_c does not contain l items do
5      if e_c contains at least one item then
6          last ← last n-best item added to e_c
7          lazy-next_n (e.cand, last)
8      if e.cand contains at least one item then
9          remove the first item from e.cand and add it to e_c
10     else
11         break
12 return e_c
```

Algorithm 5.12 ***lazy-next***$_{\mathrm{n}}$

Require: An n-best item $e_x = (r, a_x, b_y, score)$ for an equivalence class e
Require: A candidate list *cand* for e
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: The candidate list *cand* is updated

```
1  lazy-best-equiv_n (a_x, x + 1)
2  item ← (r, a_{x+1}, b_y, SCORE(r, a_{x+1}, b_y))
3  if item is not in cand then
4      add item to cand
5  lazy-best-equiv_n (b_y, y + 1)
6  item ← (r, a_x, b_{y+1}, SCORE(r, a_x, b_{y+1}))
7  if item is not in cand then
8      add item to cand
```

Configuration	LP	LR	LF	AF
Baseline	87.27	86.41	86.84	84.91
Oracle 10-best	91.50	90.49	90.99	89.01
Oracle 50-best	93.17	92.04	92.60	90.68

Table 5.3: Oracle precision (LP), recall (LR), and F-score (LF) on gold POS and F-score (AF) on automatic POS for the C&C n-best parser over CCGbank section 00.

and the process is repeated until the required number of parses are generated. ***lazy-next***$_{\text{n}}$ recursively calls ***lazy-best-equiv***$_{\text{n}}$ on the backpointers from the new candidate, propagating the lazy generation to the bottom of the chart.

Each n-best algorithm produces an equivalent n-best list for a sentence, allowing for tiebreaking procedures. Unless otherwise specified, we use Algorithm 3 for all experiments in this thesis.

5.5 Dependency Hashing

Table 5.3 gives the baseline and oracle scores for the Brennan (2008) n-best parser. We can see that on 50-best mode, the oracle score is 92.60% on gold POS tags, over a baseline of 86.84%. In contrast, the Charniak parser reaches an oracle F-score of 96.80% in 50-best mode (Charniak and Johnson, 2005). While these two metrics are calculated differently, there is a very large 4.2% difference in oracle score between the two systems. Additionally, the 7.4% deficit to a perfect F-score for CCG suggests that there may be an issue with the quality of the CCG n-best parses.

The Huang and Chiang (2005) algorithms differentiate parses by their structure, rather than the underlying logical form. To illustrate how this is insufficient for CCG, we ran the n-best C&C parser with $n = 10$ and $n = 50$, and calculated how many parses were semantically distinct (i.e. yield different dependencies). The results in Table 5.4 are striking: just 52% of 10-best parses and 34% of 50-best parses contain

	Avg. Parsers/sent	Distinct Parses/sent	Distinct %
10-best	9.8	5.1	52
50-best	47.6	16.0	34

Table 5.4: Average and distinct parses per sentence over CCGbank section 00 with respect to CCG dependencies.

	Avg. Parsers/sent	Distinct Parses/sent	Distinct %
10-best	9.8	4.4	45
50-best	47.6	13.0	27

Table 5.5: Average and distinct parses per sentence over CCGbank 00 with respect to GRs.

distinct sets of CCG dependencies. Evidently, this low percentage is a potential factor in curtailing the oracle score of the n-best C&C parser. We can also see that fewer than n parses are found on average for each sentence; this is mostly due to shorter sentences that may only receive one or two different analyses.

We perform the same diversity experiment using the C&C GR output to abstract away from the raw CCG dependencies and generate a more formalism-neutral comparison. There are even fewer distinct parses by GRs in Table 5.5: just 45% of 10-best parses and 27% of 50-best parses contain unique sets of GRs.

There is a clear mismatch between the optimisation target of the Huang and Chiang (2005) algorithms and the evaluation target for CCG. A naïve n-best CCG implementation is deficient because it does not provide sufficient differentiation in the set of dependencies generated by the parser.

5.5.1 Hashing Implementation

To address the problem of semantically equivalent n-best parses, we define a uniqueness constraint over all the n-best candidates:

Constraint. *At any point in the derivation, any n-best candidate must not have the same dependencies as any candidate already in the list.*

In an n-best dependency parser, this constraint is trivially enforced, as the analysis is defined over the dependencies. However, CCG parsing generates dependencies from category combination in a derivation. Ambiguity and equivalence mean that it is difficult to predict where in a tree dependencies will be generated, so an exhaustive comparison is required.

At its most primitive, enforcing this constraint requires comparing every dependency in one partial CCG subtree with every dependency in another subtree at every point which an n-best list is generated. This is evidently expensive. Additionally, the final set of dependencies in each n-best candidate can only be extracted during the n-best decoding step, when the optimal tree is found. Up until this point, there is no guarantee about what dependencies will be in a subtree. As parsing is already a computationally expensive process, reducing the overhead from checking this constraint is preferable.

We propose an alternative approach which represents all of the CCG dependencies in a sub-derivation using a hash value. This allows us to compare the dependencies in two derivations during decoding with a single numeric equality check rather than an exhaustive comparison. Performing this check requires only one additional integer to be stored with each n-best candidate, and one additional constant-time operation to be performed at each time two n-best candidates are considered. We call this process *dependency hashing*. In the following sections, we will describe the dependency hashing implementation, and compare it to exhaustive comparison and a fallback scheme where we use exhaustive comparison only when two hash values collide.

Huang et al. (2006) propose an idea that is similar in spirit to dependency hashing for n-best syntax-based translation reranking. They maintain a hashtable of unique strings produced at each vertex of the syntax trees driving their reranker, and as new strings are generated, any that already appear in the hash table are rejected as being

duplicates. Our technique does not use a hashtable, and instead only stores a hash value for each set of dependencies. In our task, a hash table is not suitable because dependencies form sets rather than ordered strings, and this incurs greater overhead to store in an external data structure. Our technique is faster and more memory efficient, but runs the risk of filtering unique parses due to collisions.

Using a single hash value to represent all of the dependencies present in a subtree requires three primary operations:

1) given a *filled* dependency, produce a consistent hash value;

2) given a category, produce a hash value representing all of its filled dependencies;

3) given a category and its n-th best children, combine the category's hash value with those of its children to produce a single hash value.

Recursively executing these operations from the bottom of the chart upwards will ensure that the hash value for any n-best candidate at any position represents all of the dependencies in the subtree rooted at that category.

Internally, the C&C parser stores filled dependencies as a set of four integers. In a sentence, any two dependencies sharing the same values for these integers are equivalent:

- the index of the head word of the dependency in the sentence;
- the index of the filler word of the dependency in the sentence;
- an identifier indicating the head category and filler slot;
- an identifier indicating whether the dependency originated from a given unary rule.

We can easily perform operation (1) by feeding the four integers representing a filled dependency to the C&C parser's existing hash function, a modified Bernstein hash

Algorithm 5.13 ***hash-rule***

Require: A result category r, and candidates $a_c = (a, x_a, y_a, score_a, hash_a)$ and $b_c = (b, x_b, y_b, score_b, hash_b)$, where a and b combine into r

Ensure: A hash value representing the filled dependencies in a, b, and r

1 $hash \leftarrow$ ***hash-category***(r)
2 $hash \leftarrow hash \oplus hash_a$
3 $hash \leftarrow hash \oplus hash_b$
4 **return** *hash*

Algorithm 5.14 ***hash-category***

Require: A category c from the chart

Ensure: A hash value representing the filled dependencies on the category

1 $hash \leftarrow 0$
2 **for all** filled dependencies f on c **do**
3 $\quad hash \leftarrow hash \oplus$ ***hash-dependency*** (f)
4 **return** *hash*

with a prime constant of 31.[3] This gives us a consistent hash value for all dependencies in a sentence, independent of where the dependency is generated in the chart.

Operations (2) and (3) are similar to each other, as both must effectively take two hash values representing one or more dependencies, and combine them into a single value. The key to both operations is a consistent combination operator, which can take integral hashes and combine them in an *order-independent* manner. Order independence is important as it does not matter in what order dependencies are generated in the tree. Given a set of dependencies, the final computed hash must be equal, no matter what order the dependencies are hashed. We use the exclusive OR (XOR, $\oplus$) operator for this purpose. Algorithms 5.13 and 5.14 describe the hashing operations.

We now present modifications of the Huang and Chiang (2005) algorithms incorporating our dependency hashing technique for CCG parsing. Highlighted lines have been modified to include dependency hashing.

[3] Empirical testing found this to work well (James Curran, p.c.)

Algorithm 5.15 ***hash-mult***$_{\text{n}}$

Require: Two n-best lists $a_c = a_1, a_2, ..., a_n$ and $b_c = b_1, b_2, ..., b_n$ for categories a and b, where a and b combine into a result category r
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: An n-best list r_c for r

```
1  r_c ← array
2  cand ← max-heap, sorted descending by score
3  for all a_i in a_c do
4      for all b_i in b_c do
5          score ← SCORE(r, a_i, b_i)
6          hash ← hash-rule (r, a_i, b_i)
7          add (r, a_i, b_i, score, hash) to l
8  for all c in cand do
9      (r, a_i, b_i, score, hash) ← c
10     if no item in r_c has a hash value equalling hash then
11         add c to r_c
12 return the first n elements of r_c
```

5.5.1.1 Algorithm 0

In Algorithm 0, the key change is the introduction of hash generation in the quadratic loop generating the possible n-best candidates, and the checking for hash value exclusivity in both the generation and merging steps. Algorithms 5.15 and 5.16 depict the revised procedures.

A hash uniqueness check is required every time that a candidate is considered for addition. This is necessary to ensure that the best candidates per each hash value are retained, and all other candidates with non-unique hash values are discarded. We have implemented this check by sorting the list of candidates by their scores, and iteratively adding each one as long as it has a unique hash. This allows us to remove the final sort in the base implementation. The simple iteration over the list of candidates is bounded by the small constant n representing the total number of parses required. For larger n, using a bloom filter to perform the hash equality checks will improve this performance.

Algorithm 5.16 ***hash-merge***$_{\text{n}}$

Require: Sorted n-best lists $a_c = a_1, ..., a_n$ and $b_c = b_1, ..., b_n$ for a result category r
Ensure: An n-best list r_c for r

$r_c \leftarrow$ array
while r_c does not contain n items and a_c or b_c have items **do**
    $(a, x_a, y_a, score_a, hash_a) \leftarrow a_1$
    $(b, x_b, y_b, score_b, hash_b) \leftarrow b_1$
    if $score_a > score_b$ and no item in r_c has a hash equalling $hash_a$ **then**
        remove a_1 from a_c and add it to r_c
    else if $score_b > score_a$ and no item in r_c has a hash equalling $hash_b$ **then**
        remove b_1 from b_c and add it to r_c
return r_c

5.5.1.2 Algorithm 1

Algorithm 5.17 describes the modified Huang and Chiang (2005) Algorithm 1. Each item in the candidate n-best lists now contains the hash value summarising the dependencies contained in the subtree rooted at that category. As each new candidate is generated, a new hash value is computed and stored, representing the dependencies in the new subtree under consideration. As before, candidates with non-unique hash values are discarded.

As CCG dependencies may be formed in various locations in the tree, the filled dependencies on each candidate may differ. Thus, we must still expand the successors of discarded candidates to ensure correctness.

5.5.1.3 Algorithms 2 and 3

The hashing implementation of Algorithms 2 and 3 are straightforward extensions of their non-hashing counterparts, using the principles established in the adaptations of Algorithms 0 and 1. Algorithm 2 with our dependency hashing is presented in Algorithms 5.18 and 5.19, and Algorithm 3 with hashing is presented in Algorithms 5.20 and 5.21.

Algorithm 5.17 ***new-hash-mult***$_{\text{n}}$

Require: Two n-best lists $a_c = a_1, a_2, ..., a_n$ and $b_c = b_1, b_2, ..., b_n$ for categories a and b, where a and b combine into a result category r
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: An n-best list r_c for r

```
1  r_c ← array
2  cand ← max-heap
3  add (r, a_1, b_1, SCORE(r, a_1, b_1), hash-rule (r, a_1, b_1)) to cand
4  while r_c does not contain n items and cand contains items do
5      (r, a_x, b_y, score, hash) ← the first item in cand
6      add (r, a_{x+1}, b_y, SCORE(r, a_{x+1}, b_y), hash-rule (r, a_{x+1}, b_y)) to cand
7      add (r, a_x, b_{y+1}, SCORE(r, a_x, b_{y+1}), hash-rule (r, a_x, b_{y+1})) to cand
8      if no item in r_c has a hash value equalling hash then
9          add (r, a_x, b_y, score, hash) to r_c
10     remove the first item from cand
11 return r_c
```

Algorithm 5.18 ***hash-find-best***$_{\text{n}}$

Require: An equivalence class e
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: An n-best list e_c for e

```
1  e_c ← array
2  cand ← hash-get-candidates_n(e)
3  while e_c does not contain n items and cand contains items do
4      (r, a_x, b_y, score, hash) ← the first item in cand
5      add (r, a_{x+1}, b_y, SCORE(r, a_{x+1}, b_y), hash-rule (r, a_{x+1}, b_y)) to cand
6      add (r, a_x, b_{y+1}, SCORE(r, a_x, b_{y+1}), hash-rule (r, a_x, b_{y+1})) to cand
7      if no item in r_c has a hash value equalling hash then
8          add (r, a_x, b_y, score, hash) to r_c
9      remove the first item from cand
10 return e_c
```

5.5.2 Hashing Performance

We evaluate our hashing technique with several experiments. A simple test is to measure the number of collisions that occur, i.e. where two partial trees with different dependencies have the same hash value. We parsed CCGbank section 00 with $n = 10$

Algorithm 5.19 ***hash-get-candidates***$_{\mathrm{n}}$

Require: An equivalence class e
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: A candidate list *cand* for e

```
1  cand ← heap
2  for all categories r in e do
3      a_c ← the n-best list from r's left child
4      b_c ← the n-best list from r's right child
5      score ← SCORE(r, a_1, b_1)
6      hash ← hash-rule (r, a_1, b_1)
7      if no other candidate in cand shares an identical hash then
8          add (r, a_1, b_1, score, hash) to cand
9  return cand
```

Algorithm 5.20 ***lazy-best-equiv***$_{\mathrm{n}}$

Require: An equivalence class e
Require: A limit l
Ensure: An n-best list e_c for e

```
1  e_c ← array
2  if e.cand does not exist then
3      e.cand ← get-candidates_n(e)
4  while e_c does not contain l items do
5      if e_c contains at least one item then
6          last ← last n-best item added to e_c
7          lazy-next_n (e.cand, last)
8      if e.cand contains at least one item then
9          (r, a_x, b_y, score, hash) ← the first item in e.cand
10         if no other candidate in e.cand shares an identical hash then
11             add (r, a_x, b_y, score, hash) to e_c
12         remove the first item from e.cand
13     else
14         break
15 return e_c
```

and $n = 50$ using a 64 bit hash, and exhaustively checked the dependencies of colliding states. We found that less than 1% of comparisons resulted in collisions in both 10-best and 50-best mode.

Algorithm 5.21 ***lazy-next***$_{\text{n}}$

Require: An n-best item $e_x = (r, a_x, b_y, score)$ for an equivalence class e
Require: A candidate list *cand* for e
Require: A scoring function SCORE(r, a_x, b_y) taking a_x, b_y, and result r
Ensure: The candidate list *cand* is updated

lazy-best-equiv$_{\text{n}}$ $(a_x, x+1)$
hash $\leftarrow$ ***hash-rule*** (r, a_{x+1}, b_y)
item $\leftarrow (r, a_{x+1}, b_y,$ SCORE$(r, a_{x+1}, b_y), hash)$
if *item* is not in *cand* and no candidate in *cand* shares an identical *hash* **then**
    add *item* to *cand*
lazy-best-equiv$_{\text{n}}$ $(b_y, y+1)$
hash $\leftarrow$ ***hash-rule*** (r, a_x, b_{y+1})
item $\leftarrow (r, a_x, b_{y+1},$ SCORE$(r, a_x, b_{y+1}), hash)$
if *item* is not in *cand* and no candidate in *cand* shares an identical *hash* **then**
    add *item* to *cand*

	Collisions	Comparisons	%
10-best	300	54861	0.55
50-best	2109	225970	0.93

Table 5.6: Dependency hash collisions and comparisons over CCGbank section 00.

We reran the diversity experiments, with the results summarised in Tables 5.7 and 5.8. Adding dependency hashing sees all sentences in CCGbank section 00 receive entirely unique n-best parse lists for both 10-best and 50-best settings with respect to CCG dependencies. 10-best unique parses per sentence nearly doubles from 5.1 to 9.0, and 50-best unique parses per sentence increase from 16.0 to 37.9. Overall, the total number of parses per sentence drops slightly, from 9.8 to 9.0 parses per sentence in 10-best mode and 47.6 to 37.9 for 50-best. This suggests that the hashing procedure is pruning out redundant parses, which were artificially inflating this value.

Similar results are recorded for the GR diversity (see Table 5.8), though not every set of GRs is unique due to the many-to-many mapping from CCG dependencies. Additionally, the total parse numbers for GRs are slightly different to those recorded for CCG dependencies; this is due to some slight differences in the dependency generation

	Avg. Parsers/sent	Distinct Parses/sent	Distinct %
10-best	9.8	5.1	52
50-best	47.6	16.0	34
10-best#	**9.0**	**9.0**	**100**
50-best#	**37.9**	**37.9**	**100**

Table 5.7: Average and distinct parses per sentence over CCGbank section 00 with respect to CCG dependencies. # indicates the inclusion of dependency hashing.

	Avg. Parsers/sent	Distinct Parses/sent	Distinct %
10-best	9.8	4.4	45
50-best	47.6	13.0	27
10-best#	**8.9**	**8.1**	**91**
50-best#	**37.1**	**31.5**	**85**

Table 5.8: Average and distinct parses per sentence over CCGbank section 00 with respect to GRs. # indicates the inclusion of dependency hashing.

process triggered by the use of GRs. These results show that hashing prunes away equivalent parses, creating more diversity in the n-best list.

We also evaluate the oracle F-score using dependency hashing. Our results in Table 5.9 include a 1.1% increase in 10-best mode and 0.72% in 50-best mode using the new constraints, showing how the diversified parse list contains better candidates for reranking. This demonstrates how dependency hashing addresses the deficiencies of n-best CCG parsing, and improves the quality of the resulting n-best parses. Our highest oracle F-score was 93.32% in 50-best mode.

5.5.3 Speed

Table 5.10 compares the performance of dependency hashing, dependency hashing with fallback to exhaustive checking on collisions, and exhaustive checking only across

Configuration	LP	LR	LF	AF
Baseline	87.27	86.41	86.84	84.91
Oracle 10-best	91.50	90.49	90.99	89.01
Oracle 50-best	93.17	92.04	92.60	90.68
Oracle 10-best#	**92.67**	**91.51**	**92.09**	**90.15**
Oracle 50-best#	**94.00**	**92.66**	**93.32**	**91.47**

Table 5.9: Oracle precision, recall, and F-score on gold and automatic POS tags for the C&C *n*-best parser. # denotes the inclusion of dependency hashing.

	Oracle	Alg 1	Alg 2	Alg 3
Configuration	F-score	sents/sec	sents/sec	sents/sec
baseline	86.84	44.1	44.1	44.1
		10-best		
hashing only	92.09	35.7	40.9	40.9
hashing with fallback	92.09	31.3	37.5	41.6
full comparison	92.09	2.3	15.3	40.9
		50-best		
hashing only	93.32	17.8	30.7	34.5
hashing with fallback	93.33	14.1	21.9	33.9
full comparison	93.33	0.1	1.0	25.6

Table 5.10: Oracle F-score and parsing speed in sentences per second over CCGbank 00 for Huang and Chiang (2005) Algorithms 1, 2, and 3 using dependency hashing, fallback on hashing collisions, and exhaustive dependency comparison.

three of Huang and Chiang (2005)'s *n*-best parsing algorithms on CCGbank section 00. We can see that hashing generates an identical oracle F-score on 10-best parsing, and near-identical on 50-best parsing, illustrating the correctness of the procedure, and that collisions are an infrequent issue. On the least efficient *n*-best algorithm, dependency hashing parses at 35.7 sentences per second for 10-best, and 17.8 sentences per second

for 50-best, dramatically faster than a full exhaustive dependency comparison at 2.3 and 0.1 sentences per second respectively. 10-best parsing on Algorithm 3, which requires the smallest number of dependency uniqueness checks, reveals a negligible speed difference between the hashing configurations. However, when moving to 50-best mode, dependency hashing is 33% faster than exhaustive comparison.

5.5.4 CCG reranking performance

One of the issues we found in prior work on CCG reranking was the lack of diversity in the CCG n-best parser. To address this, we ran the parser with a larger n value, and filtered out parses with identical dependencies as a post-processing step (Ng, 2010). This is clearly a poor solution, as the duplicate parses will have taken up n-best candidate slots in the parser, and potentially eliminated other analyses from being produced.

The CCG reranker is trained on data produced using a cross-fold method on CCGbank sections 02-21. Two sections are repeatedly held out, and the baseline parser is trained on the remainder of the data. Then, the parser is run in n-best mode over the held out sections, producing n-best training data. This process is repeated ten times to cover the entirety of sections 02-21, and the resulting parses are used to train the discriminative reranker. At test time, the n-best parser is first run over the test data, and then the reranker is run, reordering the parses to produce a new 1-best output.

N-best parses are required at two steps in this process: to produce the training data, and to produce the test data. To evaluate the impact of our dependency hashing algorithm in an extrinsic task, we train the Ng (2010) reranker in four configurations: with and without dependency hashing for training data and test data.

Table 5.11 shows that labeled F-score improves substantially when dependency hashing is used to create reranker training data. There is a 0.4% improvement using no hashing at test, and a 0.8% improvement using hashing at test, showing that more diverse training data creates a better reranker. The results of 87.21% without hashing at

	Test data	
Training data	no hashing	hashing
no hashing	86.83	86.35
hashing	**87.21**	**87.15**

Table 5.11: Reranked parser accuracy; labeled F-score using gold POS tags, with and without dependency hashing.

test and 87.15% using hashing at test are statistically indistinguishable from one other; though we would expect the latter to perform better.

Our results also show that the reranker performs extremely poorly using diversified test parses and undiversified training parses. There is a 0.5% performance loss in this configuration, from 86.83% to 86.35% F-score. This may be caused by the reranker becoming attuned to selecting between semantically indistinguishable derivations, which are pruned away in the diversified test set.

5.6 Summary

We have described how a mismatch between the way CCG parses are modeled and evaluated caused equivalent parses to be produced in n-best parsing. We eliminate duplicates by hashing dependencies, significantly improving the oracle F-score of CCG n-best parsing by 0.7% to 93.32%, and improving the performance of CCG reranking by up to 0.4%. Hashing is also much more efficient than exhaustive dependency comparison, while producing practically identical F-scores.

Our hashing implementation is strongly influenced by the nature of CKY-based CCG parsing: a derivation is built bottom-up, with category combination generating dependencies. Unlike the dependency parsers in Chapters 3 and 4, which build trees as sets of dependencies, working with dependencies in CCG is complicated by the fact that equivalent derivations can yield the same dependencies in different locations.

Even determining the final set of dependencies in a CCG analysis cannot be done until the chart is decoded, rather than while the possible derivations are being built.

In the next chapter, we implement a constraint-based evaluation for the C&C parser, based on the procedure developed in Chapter 3. A theme of this chapter has been that CCG parsing is complex, and algorithms which apply naturally to other formalisms require extra care for CCG. In particular, enforcing properties of the sets of dependencies generated by the parse is a nontrivial task. We shall see this theme continue in our constraint evaluation implementation for C&C.

6 Dependency Constraints for CCG

In this chapter, we return to the notion of dependencies as constraints from Chapter 3, and implement a constraint-based evaluation procedure for the C&C CCG parser. We use the formalism-independent grammatical relations output of the parser to develop a group of error classes over CCG dependencies, similar to the classes used for our dependency parsing experiments. Using the dependencies in each error class, we force the C&C parser to produce derivations containing the constrained dependencies, and analyse the implications of applying constraints throughout the remainder of the parse.

The CCG parsing process is complex. Unlike dependency parsers, which assign a single head to each word in a sentence, and thus can directly model and construct head-argument pairs, CCG dependencies are a representation of the logical form generated by a derivation. They are built during parsing as the product of a unification-based process, driven by co-indexation and slot annotations on CCG categories. The previous chapter described the challenges in applying properties over the set of dependencies generated by a parse, and in this chapter we describe the intricacies of enforcing dependency existence. We provide the parsing and decoding algorithms necessary for robustly applying these existence constraints.

This chapter is arranged as follows. First, we examine the complexities of enforcing dependency constraints in CCG. We extend the CKY CCG parsing algorithm of Steedman (2000) and the normal-form C&C decoding algorithm of Clark and Curran (2007b) to implement constraint application. Finally, we discuss how to group CCG dependencies into error classes, and note the intricacies of extracting coordination and

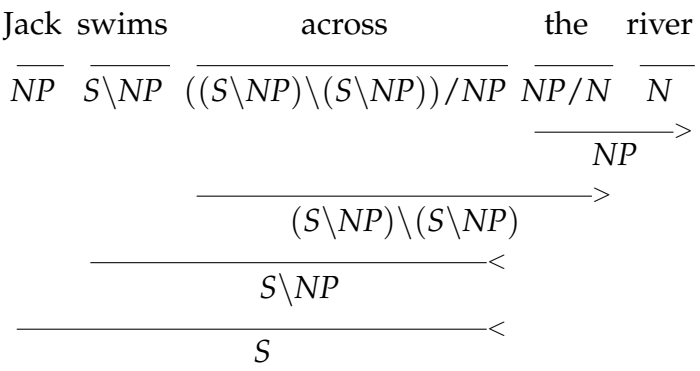
root constraints from CCG derivations. The next chapter uses the constraints from this chapter for evaluating and exploring the C&C CCG implementation.

6.1 CCG Constraints

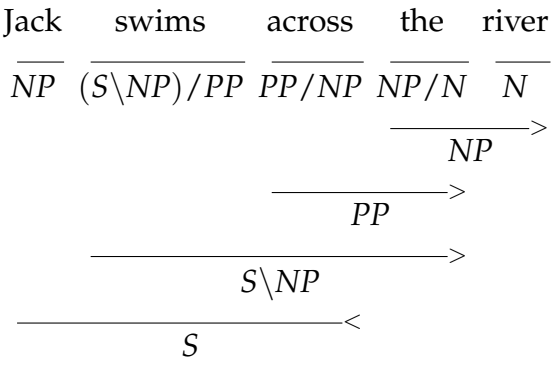
In the CCG dependency recovery metric, all components of a dependency must match the gold standard for it to be scored as correct, including the category assigned to the head word, and the slot which the argument word fills. This makes the metric much stricter than PARSEVAL. In Parse 6.1, the PP *across the river* has been interpreted as an adjunct, rather than an argument as in Parse 6.2. Both parses would score identically under PARSEVAL as their bracketing is unchanged. However, the adjunct to argument change results in different categories for *swims* and *across*, as the verb category $(S\backslash NP)/PP$ now subcategorises for the required PP. Whilst every other category in the sentence has remained the same, nearly every dependency in the sentence is headed by *swims* or *across*, and thus each dependency changes as a result. An incorrect argument/adjunct distinction in this sentence produces an F-score of only 25% even though there are only two categories different between the options.

In Parse 6.1, there is a CCG dependency headed by *across* and filled by *swims*, representing the modifier application. Parse 6.2 reverses the direction of the dependency, as *swims* subcategorises for the PP. Intuitively, enforcing the correct category, slot, and direction for this dependency would correct the errors resulting from an incorrect argument-adjunct distinction.

Unlike the dependency parsers examined in Chapters 3 and 4, CCG has no restriction on the number of heads that a particular word may have. Chapter 5 described how derivations can yield the same sets of dependencies, despite being structurally different, and how the final set of dependencies in a CCG analysis is not finalised until the chart is decoded. Thus, enforcing the appearance of certain dependencies in the final CCG parse is substantially more complex than in dependency parsers. Subtrees cannot simply be



Parse 6.1: A CCG derivation with a PP adjunct, adapted from Villavicencio (2002).



Parse 6.2: A CCG derivation with a PP argument (note the categories of *swims* and *across*). The bracketing is identical to Parse 6.1, but nearly all dependencies have changed.

ruled out because a word has been assigned a different head to the one expected in a constraint. There is no restriction on where in a CCG derivation a particular dependency will form, unlike in dependency parsers where an attachment between a head and its argument must occur when the arc between them is being considered by the parser.

CCG dependencies are formed as categories are combined together and their variables are unified. In the C&C parser, *active variables* carry the words of the sentence which are available for dependencies to accept as heads or fillers. Thus, given a constraint specifying a desired CCG dependency, we can identify two situations when the dependency is impossible to create, namely observing that:

1) the head word of the dependency does not bear the required category;

2) the filler word of the dependency becomes inactive before the dependency has been created.

A CCG dependency may be created at any point in the subtree which contains the head word (with the correct category) and the filler word as leaf nodes. However, in this subtree, the filler word must be passed upwards as an active variable until it can combine with the complex category to create the dependency. If at any point the filler word becomes an inactive variable (e.g. it is absorbed by another category which does not allow it to be further used, or it has the wrong category itself to allow the variable to be unified correctly), then the dependency cannot possibly be created. This observation forms the basis of our technique to enforce dependency-level constraints in CCG parsing.

Given a list of constraints, we store each constraint on the expected head category in the bottom row of the chart, and mark any non-satisfying categories. Then, as we are decoding the packed chart, we propagate constraints and the satisfying status of categories upwards. At each combination, the result category is immediately marked as non-satisfying if either one of its children are non-satisfying. Otherwise, we inspect the newly created dependencies on the result category to see if any constraints on its children have been satisfied. We also verify that no required variable has become inactive when two categories have been combined; if this has occurred, we mark the result category as not satisfying. Finally, any unsatisfied constraints are then stored on the result category.

Once we have decoded the chart and propagated the constraints and satisfying properties through all spanning analyses, we return the root node of the tree with the highest score under the parser model that satisfies all of the applied constraints. It is worth noting that unlike earlier experiments on dependency constraints, the C&C parser is still building all alternative parses, and we eliminate those which do not satisfy the constraints in decoding.

Algorithm 6.1 ***load-constraints***

Require: A chart *chart* for a sentence, with span 1 cells populated
Require: A list of consistent constraints (*head_index*, *cat*, *filler_index*, *slot*)
Ensure: The constraints are loaded onto the appropriate categories

```
1  for all cells cell in chart with span 1 do
2      pos ← position of the cell
3      for all categories c in cell do
4          required ← all constraints cons where cons.head_index == pos
5          correct_cat ← the head category from any cons in required
6          if c == correct_cat then
7              add required to the list of constraints on c
8              mark c as satisfied
9          else
10             mark c as unsatisfied
```

Unlike Djordjevic (2006), who also experiment with adding constraints to the C&C parser, our constraints operate directly over CCG dependencies, rather than constituents in the CCG derivation. The spurious and attachment ambiguity inherent in CCG means that several constituent structures may yield the same set of dependencies, so our dependency constraints abstract over many equivalent derivations. This aligns our procedure more closely with the underlying logical form produced by the formalism.

6.2 Implementation

Constraints are represented as 4-tuples $\langle h, c, s, f \rangle$, similar to CCG dependencies (see Section 2.3.3). h and f are the head and filler of the dependency, c is the category assigned to the head word, and s is the slot filled by f.

Our dependency-level constraint implementation in the C&C parser has two main phases. Firstly, as the categories from the supertagger are loaded into the chart, we iterate through all cells with span 1, adding the constraints to the appropriate category in the appropriate cell. In each cell, we also mark any non-conforming categories as being non-satisfying. This procedure is described in Algorithm 6.1.

Algorithm 6.2 ***decode-constrained***$_{\text{nf}}$

Require: A packed chart *chart* with constraints loaded
Ensure: Returns the best scoring tree root satisfying the constraints

```
1  max_e ← ∅
2  max_s ← −∞
3  max_sat ← 0
4  for all categories c in the cell of chart spanning the whole sentence do
5      equiv, score, nsatisfied ← best-equiv-constrained (c)
6      if equiv is marked as satisfying then
7          if max_sat < nsatisfied or (max_sat == nsatisfied and max_s < score) then
8              max_e ← equiv
9              max_s ← score
10             max_sat ← nsatisfied
11 return max_equiv
```

The primary component of our procedure occurs during decoding, when the chart has been completely built and the best scoring spanning analysis of the sentence is found. As decoding proceeds, we propagate constraints upwards from the bottom of the chart, checking them off as they are satisfied, and marking categories as non-satisfying if they violate any constraints or contain a non-satisfying child. Algorithms 6.2 to 6.6 describe constraint-based decoding, adapting the procedure described in Section 5.2.3.

Algorithm 6.3 ***best-equiv-constrained***

Require: an equivalence class e

Ensure: the category in e with maximal score and number of satisfied constraints

```
1  if e.max_equiv exists then
2      return e.max_equiv, e.max_score, e.max_nsatisfied
3  max_e ← ∅
4  max_s ← −∞
5  max_sat ← 0
6  for all categories c in e do
7      score, nsatisfied ← best-score-constrained (c)
8      if c is marked as satisfying then
9          if max_sat < nsatisfied or (max_sat == nsatisfied and max_s < score) then
10             max_e ← c
11             max_s ← score
12             max_sat ← nsatisfied
13 e.max_equiv ← max_e
14 e.max_score ← max_s
15 e.max_nsatisfied ← max_sat
16 return max_e, max_s, max_sat
```

Algorithm 6.4 ***best-score-constrained***

Require: a category c in the chart
Ensure: the best score for the subtree rooted at c
Ensure: the number of constraints satisfied in the best subtree rooted at c
Ensure: all unsatisfied constraints in c's children are propagated to c
Ensure: c is marked as not satisfying all constraints if a violation occurs

```
score ← SCORE(c)
nsatisfied ← 0
mark c as satisfying
if c has a left backpointer c.left then
    max_left, s, left_satisfied ← best-equiv-constrained (c.left)
    score ← score + s
    nsatisfied ← nsatisfied + left_satisfied
    if max_left is not marked as satisfying then
        mark c as not satisfying
    if c has a right backpointer c.right then
        max_right, s, right_satisfied ← best-equiv-constrained (c.right)
        score ← score + s
        nsatisfied ← nsatisfied + right_satisfied
        if max_right is not marked as satisfying then
            mark c as not satisfying
if c is marked as satisfying then
    nsatisfied ← nsatisfied + check-satisfaction (c, max_left)
    nsatisfied ← nsatisfied + check-satisfaction (c, max_right)
    copy unsatisfied constraints from max_left and max_right to c
    if violates-variables (c, max_left) or violates-variables (c, max_right) then
        mark c as not satisfying
return score, nsatisfied
```

Algorithm 6.5 ***check-satisfaction***

Require: a category c
Require: a category p that is a child of c
Ensure: the number of constraints from p that are satisfied in r

```
1 nsatisfied ← 0
2 for all constraints cons in p do
3     for all filled dependencies f in c do
4         if f matches cons then
5             nsatisfied ← nsatisfied + 1
6             mark cons as satisfied
7 return nsatisfied
```

Algorithm 6.6 ***violates-variables***

Require: a category c
Require: a category p that is a child of c
Ensure: true if any constraint on c requires a variable that was active in p and no longer active in c; false otherwise

```
1 for all constraints cons in c do
2     if the filler word of cons fills any active variable v in p then
3         if the filler word of cons is no longer an active variable in c then
4             return true
5 return false
```

The constraint checking procedure requires several indicator flags and counters to be cached on categories in the chart. It may, and often does, reject spanning analyses for not satisfying or violating the supplied constraints. This has the side effect of breaking the assumption underpinning the chart repair procedure described in Section 5.2.5, where cached data from a parse attempt remains valid for any subsequent reattempts. In turn, on subsequent parse attempts, the parser reuses previously computed constraint and satisfaction values, and does not recompute them based on a new category available in the same equivalence class. We counteract this by resetting the indicator flags and constraint counters if the parser cannot find a spanning analysis, and reattempts with additional categories. Importantly, this does not change the chart repair process, and preserves the parser's original behaviour.

6.3 Creating Constraints

Similar to our investigation for dependency parsers in Chapter 3, we define several error classes based on common kinds of difficult constructions for parsers. In this section, we describe how those classes are defined.

6.3.1 Error Classes

We use the CCG dependency to GR mapping developed by Clark and Curran (2007a) to perform a formalism-independent evaluation of the C&C parser. The mapping transforms CCG categories into GRs, abstracting CCG categories and slots into meaningful labels such as *subject* and *object*. The abstraction reduces the number of head-argument relationships that we have to consider from thousands to just under twenty. It also makes the task similar to the mapping procedure described in Section 3.5.2.

Rimell and Clark (2008) and Rimell et al. (2009) describe the development of a CCG dependency to Stanford dependency mapping, implemented using the same techniques as the GR mapping. However, Stanford dependencies differ even more from

CCG than do GRs, and the conversion process requires substantial post-processing to align the dependencies from the parser more closely to Stanford dependencies. We did not make use of this mapping due to these issues.

We define error classes for CCG based on the GR label descriptions from the RASP Parser Technical Report (Briscoe, 2006). There are fewer labels in the GR hierarchy compared to the Stanford dependency scheme, making it more difficult to differentiate between certain error classes. In particular, there are fewer distinctions between different nominal behaviours, such as appositions, modifiers and possessives. There are also fewer distinctions between phrasal types; the *ncmod* relationship encompasses the vast majority of pre- and post-modifiers, including adjectival, adverbial, nominal, and prepositional. This means that additional heuristics based on the filler word's POS tag must be used to partition the dependencies into error classes.

Our error classes are constructed to match those in Section 3.5.2 as closely as possible, allowing for a broader comparison of the CCG parser with the dependency parsers. The classes are defined as follows:

PP attachment: any label attaching a prepositional phrase. Includes *pmod*, *iobj*, and *pcomp*. Also includes *ncmod*, *ncsubj*, and *iobj* if the POS tag of the filler is TO or IN;

NP internal: any label marking nominal structure (not including adjectival modifiers). Includes *det*. Also includes *ncmod* if the GR attributes include *poss*, *part*, or *num*, or if the POS tag of both head and filler word is one of NN, NNS, NNP, NNPS, PRP, PRP$, DT, $, CD, or #;

NP attachment: any label specifically attaching an NP. Includes *ncsubj*, *dobj*, *obj*, and *obj2*;

Clause attachment: any label attaching a clause. Includes *cmod*, *xsubj*, *csubj*, *xcomp*, and *ccomp*;

Modifier attachment: any label attaching an adverbial or adjectival modifier. Includes *xmod*. Also includes *ncmod* if it does not satisfy the requirements of NP attachment;

Other attachment: all other cases, specifically *aux*, *ta*, *arg*, and *arg_mod*.

Unlike dependency grammars which specify that every word must have a head, C&C does not include dependencies for punctuation attachment. Instead, punctuation is commonly absorbed into larger constituents without any impact on the logical form or on the category of the absorbing constituent (see Section 2.3.4 for more details). Punctuation absorption primarily impacts the parser's speed and coverage; as punctuation may be absorbed in many locations, it contributes to the chart size growing to its predefined limit. Djordjevic (2006) found that basic span-level constraints forcing sentence-final punctuation to attach at the root of the derivation, and semicolon constraints splitting the sentence into smaller chunks were advantageous for parsing speed and coverage for this reason.

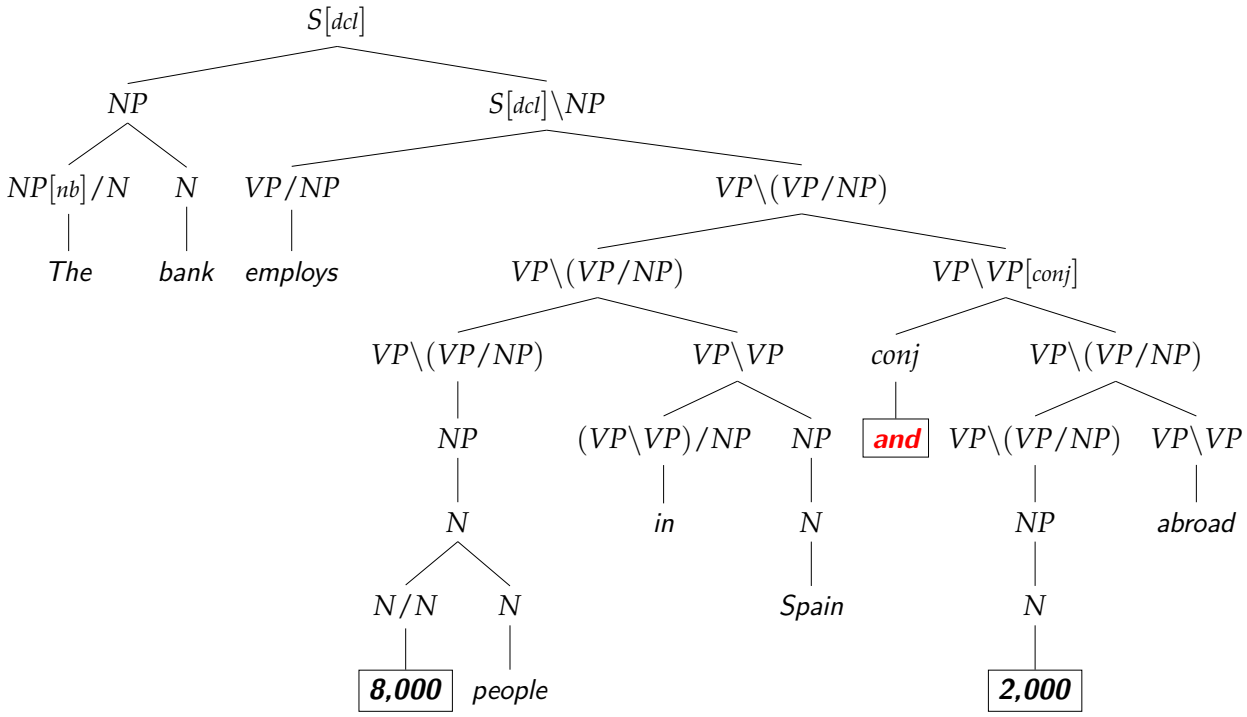
6.3.2 Constraints from CCGbank

We use the gold-standard CCGbank dependencies as the source of constraints. For each dependency, we map it to its equivalent GR, and then place the dependency in the error class defined above based on its GR.

When evaluating the parser using a certain error class, the dependencies which have been placed in that class are activated and enforced for each sentence in CCGbank section 00. Notably, we do not have *conj* or *root* types listed in the error classes above. This is due to the special way in which these dependencies are handled in CCG. We describe below how we use the CCGbank derivations, rather than the gold-standard dependencies, to extract constraints for these error classes.

Coordination Constraints

CCGbank does not include *conj* dependencies, and the *conj* output produced by the parser is ignored during the standard evaluation. However, the normal-form derivations provided by the corpus do include *conj* nodes. Each *conj* node in the derivation generates two dependencies: one with the head of the left input category to the *conj*, and one with the head of the right input category. In CCGbank, which binarises the



Parse 6.3: A CCGbank tree containing coordination; where the *conj* and two filler words are boxed. The transitive verb category $S[dcl]\backslash NP$ is shortened to VP.

ternary coordination rule, the left input category corresponds to the head of the left child of the grandparent of the *conj* node; the right input category is the head of the right child of the parent of the *conj* node. Parse 6.3 depicts the relationship in the tree between the nodes in question, and Algorithm 6.7 gives the extraction process.

Care must be taken in list sentences, where multiple items are coordinated together. In this case, each *conj* node coordinates *all* heads of its two input categories, where each conjunction creates a category with two heads.

Once we have extracted the additional *conj* dependencies from CCGbank section 00, we add them to the gold standard for each sentence. They are applied as constraints using the same algorithm as described in Section 6.2, as the C&C parser does generate these dependencies.

Algorithm 6.7 ***extract-conj-deps***

Require: a CCGbank derivation T
Ensure: constraints specifying all *conj* dependencies in the derivation if they exist

```
constraints ← empty list
for all leaf nodes node in T do
    if node has a parent and that parent has category matching *[conj] then
        left_heads ← the set of heads of the left child of node's grandparent
        right_heads ← the set of heads of the right child of node's parent
        for all head in left_heads, right_heads do
            add a conj constraint between node and head to constraints
return constraints
```

```
(<T S[dcl] 0 2>
  (<T S[dcl] 1 2>
    (<T NP 1 2>
      (<L NP[nb]/N DT The>)
      (<L N NN bank>) )
    (<T S[dcl]\NP 0 2>
      (<L (S[dcl]\NP)/NP VBZ employs>)
      (<T NP 0 1>
        (<T N 1 2>
          (<L N/N CD 8,000>)
          (<L N NNS people>) ) ) ) )
  (<L . . . . .>) )
```

Figure 6.1: An CCGbank tree with the path to the root word following the head annotations on each node bolded.

Root Constraints

CCG has no concept of an external explicit root node, like many dependency schemes. This precludes an explicit *root* dependency. However, the headedness information in CCGbank derivations identifies one leaf node which heads the entire tree based on the head-finding heuristics employed by Hockenmaier (2003a). Figure 6.1 gives an example tree from CCGbank, highlighting the nodes on the path to the root leaf node indicated by the headedness annotations. Algorithm 6.8 describes how we extract root constraints from these trees.

Algorithm 6.8 ***extract-root-deps***

Require: a CCGbank derivation T
Ensure: a constraint specifying the root of the derivation

```
1 node ← the root of T
2 while node is not a leaf do
3     if node is headed by its left child then
4         node ← node.left
5     else
6         node ← node.right
7 return the index position of node in the sentence
```

Algorithm 6.9 ***apply-root-constraints***

Require: a filled chart C
Require: the index position of the root in the sentence r
Require: the category of the root in the sentence R
Ensure: roots of the chart without the required root word active are marked as not satisfying constraints

```
1 for all equivalence classes e in the root cell of C do
2     for all categories c in e do
3         if r is the active variable X in c and c == R then
4             mark c as satisfying
5         else
6             mark c as not satisfying
```

The head word for a sentence from CCGbank corresponds to the active variable X on the root categories of the chart in the C&C parser. Extracting the head of each sentence in CCGbank section 00 gives an approximation of a root constraint, but we cannot enforce it in the same way as our other constraints since the root will not necessarily correspond to a single dependency. The root word of a sentence may have multiple modifiers, and itself head several other dependencies. Instead, root constraints are enforced by inspecting the active variables in the root cells of the chart, and only allowing those where the desired word is in the active X variable. Additionally, any trees where the category of the root word is incorrect are rejected. Algorithm 6.9 describes this procedure, which is appended to the end of Algorithm 6.2.

6.3.3 Constraint Statistics

The CCG to GR mapping is only defined over the 425 lexical categories used by the C&C parser (Clark and Curran, 2007a). Additionally, the parser generates some dependencies which do not have an equivalent GR. This is particularly prevalent in CCG subject dependencies. For example, the $S[adj]\backslash NP$ category, used to represent adjectival phrases, is a synthetic category. The argument NP is necessary to allow adjectives to pick up their subject NPs in copula constructions, but this dependency does not have an equivalent GR. These extraneous dependencies are ignored by the mapping, and so using GRs to define our error classes is deficient. Table 6.1 gives the number of dependency constraints placed into each error class by our procedure over CCGbank section 00, as well as those which do not have an equivalent GR, or are ignored in the mapping. It also gives the distribution over the Stanford dependency error classes used in Chapter 3.

There are 46,099 total dependency instances in CCGbank section 00, including the *root* and *conj* constraints extracted directly from the derivations. 159 instances do not have an entry in the mapping, while 2,427 are ignored. All of the remaining dependencies are mapped into a GR and thus into an error classes.

While the constraints are extracted over different sections of newswire, there is a very similar proportion of root attachments across CCGbank and Stanford dependencies, indicating that sentence lengths are broadly similar. NP-related attachments are most frequent across both sets, though CCGbank has substantially more of both NP attachment and NP internal. Meanwhile, there is a large discrepancy in the Other attachment set. For CCGbank, this class is composed almost entirely of *aux* relations between auxiliaries and verbs in sentences. This is in contrast to Stanford; as every word requires a head and label, there are many more labels which would not receive an equivalent GR relation, and thus placed in the Other class.

Constraint Type	Freq.	%	Stanford %
NP attachment	13768	29.9	21.2
NP internal	10070	21.8	20.8
Modifier attachment	5339	11.6	13.3
PP attachment	4732	10.3	9.3
Clause attachment	3650	7.9	4.2
Other attachment	259	0.6	10.4
Coordination attachment	3782	8.2	5.1
Root attachment	1913	4.1	4.2
Not mapped	159	0.3	-
Ignored	2427	5.3	-
Total	46099	100.0	100.0

Table 6.1: The number of constraints in each error class over CCGbank section 00, compared to the distribution of constraints per Stanford error class over OntoNotes WSJ section 22 in Section 3.5.2.

Stanford also has slightly more modifiers, and fewer clauses and coordination than CCGbank. Each instance of coordination in CCGbank yields two dependencies, and coordinated lists add a compounding number of dependencies of this class. In Stanford, additional coordinated items add only two additional constraints, explaining some of the difference in relative size.

6.4 Summary

In this chapter, we have taken our constraint-based evaluation procedure for dependency parsers and described how to implement it for the C&C CCG parser.

Unlike the relatively simple implementation for dependency parsers, enforcing constraints in C&C is challenging. As we found in the previous chapter, working with CCG dependencies during parsing is difficult, as they are a by-product of category combination in a derivation. In dependency parsers, we could simply prevent undesired

dependencies from being formed as the tree is constructed. However, in CCG we cannot know the dependencies present in the final analysis until decoding, meaning that applying constraints must be deferred until after the chart is constructed. Our implementation propagates constraints bottom-up through the chart during decoding, checking for violations at each combination, and summing the total number of satisfied constraints.

We used the formalism-independent GR output of the C&C parser to map CCG dependencies to error classes mimicking those in Chapter 3. We also describe how to extract coordination and root relations directly from CCGbank derivations, as they are not included with the dependencies in the corpus. The deficiency of the CCG to GR mapping is challenging for producing error classes: GRs are less specific than the Stanford labels used for our dependency parser evaluation, and not all CCG dependencies map to GR labels. Nonetheless, we were able to map over 94% of the dependencies in CCGbank section 00 to an error class, with the overall distribution being broadly similar to Stanford. CCG has substantially more NP attachments, clauses, and coordination, but fewer modifiers and PPs, traits explained by the particulars of how CCG treats these constructions.

In the next chapter, we will use the constraint implementation and error classes to perform an in-depth exploration of the C&C parser's performance, comparing it to the dependency parsers in Chapter 3.

7 Evaluating with CCG Constraints

In this chapter, we apply the constraint-based CCG evaluation procedure defined in the previous chapter, and perform an in-depth investigation of the C&C parser's accuracy across different error classes, comparing it with the dependency parsers in Chapter 3.

We find that many of the substantial error classes for dependency parsers are also challenging for CCG, including NP and PP attachments, and clauses. However, the impact of each error class is different between the formalisms. Particularly notable is the reversal of error distribution for NP and PP attachments. The former causes similar amounts of constrained and cascading impact in dependency parsers, but proportionally less cascading impact for CCG. PPs exhibit the opposite behaviour: less cascading impact in dependency parsers, but more in CCG. Coordination can also cause negative cascading impact in CCG, while it has a strictly positive overall effect for dependency parsers. These differences are due to the way each formalism handles these constructions; the head category sometimes changes for misattached PPs, as well as the lack of enforcement for correct structure in the arguments of coordination explain the behaviour of these error classes in CCG.

Our constraint-based procedure can also be used to evaluate a CCG parser and its implementation in detail, highlighting inconsistencies in the co-indexation annotations and grammar implementation. Applying constraints can have unexpected side-effects in CCG due to the interaction between the subcategorisation encoded in CCG categories, and the application of CCG unary rules and combinators. We present several examples

where the C&C parser's response to constraints is unusual, demonstrating the use of our procedure in parser debugging.

This chapter is arranged as follows. We detail an evaluation procedure which partitions constrained parses into dependencies which have been directly and indirectly affected by the constraints. Finally, we run our evaluation procedure over the error classes, and discuss the results and implications for CCG parsing.

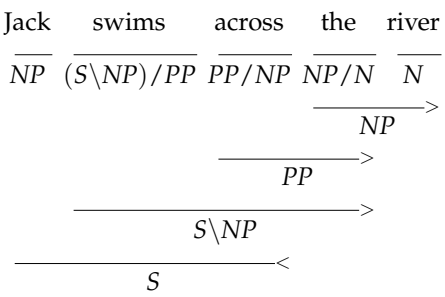
7.1 Evaluation Procedure

Unlike our constraint evaluation procedure for dependency grammars in Chapter 3, evaluating the effectiveness of CCG constraints is a nontrivial task. The key issue is that applying a constraint in a CCG parse may achieve any, several, or all of the following:

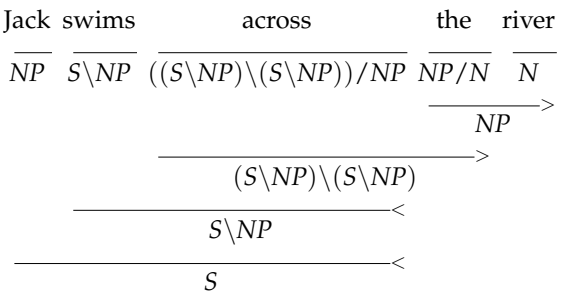
- change a single incorrect dependency to a correct dependency;
- add a dependency where none existed before;
- remove an existing dependency.

Constraints may, and often do, create a parse with a different number of dependencies than the baseline. This precludes the simple partitioning of each parse into *constrained* and *cascaded* dependency sets, as is possible for our dependency parser experiments in Chapter 3. Additionally, calculating the CCG F-score metric requires the full set of dependencies for a sentence, as it relies not only on the total number of correct dependencies recovered by the parser, but also the number of incorrect and missing dependencies according to the gold standard. In contrast, the accuracy metric used for dependency grammars with a single head per word may easily decomposed into constrained and cascaded components as it is a simple ratio.

In this section, we describe a procedure to construct constrained and cascaded dependency sets that are evaluable against the gold standard, given the baseline C&C parser output, and the parser output with constraints applied.



Parse 7.1: A baseline CCG derivation with a PP argument.



Parse 7.2: A constrained CCG derivation with a PP adjunct.

Trivially, both constrained and cascaded dependency sets must contain all dependencies in common between the baseline and constrained parses. These must be already correct in the baseline, or left unchanged by applying constraints. Of the remaining dependencies in the constrained parse, those which directly correspond to a constraint must belong to the constrained set, while those that do not go in the cascaded set.

The key challenge is identify where the changed dependencies from the baseline parse should be placed. Clearly, incompatible dependencies should be placed in different sets to allow a complete evaluation to take place. Consider an example where Parse 7.1 is a baseline parse of the sentence, and Parse 7.2 is the result of applying a constraint enforcing the dependency between *across* and *swims*. Figure 7.1 lists the dependencies generated by each derivation. We can immediately identify a *correspondence* between the dependencies on each line. On the first line, the only difference in the dependencies is a change of head category. The third line has dependencies with a different head category and slot, but the same head and filler. The second line has dependencies where the direction is reversed. The fourth line has two identical dependencies.

This notion of correspondence allows us to heuristically match pairs of changed dependencies. When the dependency on one side is placed in the cascaded set, the corresponding dependency must be placed in the constrained set, and vice-versa. The partitioning procedure proceeds as thus:

$\langle$ *swims*, $(S[dcl]\backslash NP_1)/PP_2$, 1, *Jack*, $-\rangle$	$\langle$ *swims*, $S[dcl]\backslash NP_1$, 1, *Jack*, $-\rangle$
$\langle$ *swims*, $(S[dcl]\backslash NP_1)/PP_2$, 2, *across*, $-\rangle$	$\langle$ *across*, $(S\backslash NP)_1\backslash(S\backslash NP)/NP_2$, 1, *swims*, $-\rangle$
$\langle$ *across*, PP/NP_1, 1, *river*, $-\rangle$	$\langle$ *across*, $(S\backslash NP)_1\backslash(S\backslash NP)/NP_2$, 2, *river*, $-\rangle$
$\langle$ *the*, $NP[nb]/N_1$, 1, *river*, $-\rangle$	$\langle$ *the*, $NP[nb]/N_1$, 1, *river*, $-\rangle$

(a) Baseline parser dependencies. **(b)** Constrained parser dependencies.

Figure 7.1: Dependencies yielded by the baseline parse of Parse 7.1 and constrained parse of Parse 7.2.

- all dependencies in the constrained parse matching an applied constraints are added to the *constrained* set. Any *corresponding* dependencies in the baseline parse are added to the *cascaded* set;
- all dependencies in the constrained parse which do not match a constraint are added to the *cascaded* set. Any *corresponding* dependencies in the baseline parse are added to the *cascaded* set;
- any remaining dependencies in the baseline parse are added to the *cascaded* set.

Comparing the F-score of these dependency sets against the baseline gives us a close approximation of the F-score improvement that is directly due to the constraints and indirectly due to the constraints.

There are several levels of strictness in dependency correspondence, due to the number of ways that applying constraints can change the parser output. These include:

1) a head category change;

2) a reversal of the head and filler;

3) a head category and slot change;

4) a head category and filler change;

5) a head category, slot, and filler change.

When determining dependency correspondence, we iteratively check all remaining changed dependencies at each level, reducing the strictness required each time. This means that as many correspondences are determined with the highest level of strictness, whilst allowing fallback to more lenient heuristics. Algorithm 7.1 describes the full partitioning procedure.

Figure 7.2 lists the partitioned constrained and cascaded dependency sets created using our procedure from Figure 7.1, with corresponding dependencies depicted next to one another. The constrained and cascaded output are not internally consistent; *across* and *swims* head dependencies with different categories on the constrained and cascaded side respectively. However, the intention of the partitioning is not to create real parser output; we merely wish to divide the changes produced by applying constraints into two representative sets of cascading and constrained dependencies which can be evaluated using the usual CCG dependency recovery metric.

The total difference in F-score between the baseline and constrained parsers is denoted as $\Delta\mathbf{LF}$. Using the partitioning output, we define Δc, the difference between the baseline parse and the constrained dependency set, and Δu, the difference between the baseline parser and the cascaded dependency set. Ideally:

$$\Delta\mathbf{LF} = \Delta c + \Delta u \tag{7.1}$$

Despite the partitioning not being perfect, and the intricacies of calculating an F-score precluding this simple process from properly mimicking the precision of using accuracies in Chapter 3, empirical testing and the results presented in this chapter show that partitioning is more than adequate for our evaluation. Equation 7.1 is satisfied within a margin of error of a few hundredths of a percent for all of our experiments.

Algorithm 7.1 ***make-dep-sets***

Require: the dependencies produced by the baseline unconstrained parser B
Require: the dependencies produced by the constrained parser T
Require: the list of constraints l applied to the constrained parser
Ensure: a *constrained* and *cascaded* set of dependencies for evaluation

```
1  relax ← 0
2  constrained ← empty set
3  cascaded ← empty set
4  common ← B ∩ T
5  base_remaining ← B \ common
6  test_remaining ← T \ common
7  while there are dependencies in test_remaining do
8      for all dependencies d in test_remaining do
9          corresponding ← find-corresponding(base_remaining, d, relax)
10         if d matches a constraint in l then
11             add d to constrained
12             if corresponding is not null then
13                 add corresponding to cascaded
14                 remove d from test_remaining
15                 remove corresponding from base_remaining
16         else
17             add d to cascaded
18             if corresponding is not null then
19                 add corresponding to constrained
20                 remove d from test_remaining
21                 remove corresponding from base_remaining
22     relax ← relax + 1
23 cascaded ← cascaded ∪ base_remaining
24 return constrained, cascaded
```

$\langle$ *swims*, $(S[dcl]\backslash NP_1)/PP_2$, 1, *Jack*, $-\rangle$

$\langle$ *across*, $(S\backslash NP)_1\backslash(S\backslash NP)/NP_2$, 1, *swims*, $-\rangle$

$\langle$ *across*, PP/NP_1, 1, *river*, $-\rangle$

$\langle$ *the*, $NP[nb]/N_1$, 1, *river*, $-\rangle$

(a) *Constrained* dependency set.

$\langle$ *swims*, $S[dcl]\backslash NP_1$, 1, *Jack*, $-\rangle$

$\langle$ *swims*, $(S[dcl]\backslash NP_1)/PP_2$, 2, *across*, $-\rangle$

$\langle$ *across*, $(S\backslash NP)_1\backslash(S\backslash NP)/NP_2$, 2, *river*, $-\rangle$

$\langle$ *the*, $NP[nb]/N_1$, 1, *river*, $-\rangle$

(b) *Cascaded* dependency set.

Figure 7.2: The partitioned constrained and cascaded dependency sets generated by Algorithm 7.1 from Figure 7.1. Corresponding dependencies are next to each other. Each set is not internally consistent (e.g. the second and third constrained dependencies), but this is not a goal of partitioning.

7.2 Experimental Setup

We use the same settings, model, and data for the C&C POS tagger, supertagger, and parser as described in Section 5.3.

The additional complexity of the CCG parsing process and the nature of CCG dependencies as constraints means that there are several scenarios where the parser may be unable to satisfy the constraints and produce a parse. Some of these situations are due to the design of the C&C parser, while others are due to the way in which CCG rules are implemented in CCGbank. We experiment with a number of relaxations to the constraints to investigate their impact on the parser's coverage and accuracy:

Strict enforcement: all constraints for each sentence must be fully satisfied. A sentence without a spanning analysis satisfying all provided constraints is rejected as unparseable. This is the strictest configuration.

Add missing: some constraints are unsatisfiable because the supertagger does not provide the required head categories. In this configuration, missing head categories are added to the C&C chart before parsing begins. This alters the parser's behaviour, but isolates errors made by the supertagger from inadequacies of the grammar implementation in the parser.

Any category or slot: relax constraint enforcement to be word-word only, and ignore the category and slot requirements. This configuration simply requires that any CCG dependency exist between the head and filler provided in a constraints, rather than enforcing a full category and slot match. This configuration relies on the supertagger providing the necessary correct categories, and the parser being able to select them as the best option. However, any dependency between the head and filler word is permitted, introducing the potential for errors.

Constraints at root: permit sentences which only receive a spanning analysis that does not satisfy all the required constraints. This addresses cases where the parser constructs the desired derivation, but particulars of the grammar implementation

prevent a constraint dependency from being correctly formed. However, analyses which violate a constraint are still rejected.

We address sentences which do not receive an analysis under these four configurations in two different ways:

Reject on failure: discard sentences which do not receive an analysis under constraints as unparseable.

Reparse on failure: fall back to the baseline parser if a sentence does not receive an analysis under constraints.

7.2.1 Configuration Examples

Parse 7.3 gives the C&C analysis and gold standard parse of a CCGbank section 00 sentence. The parser has chosen an incorrect category for *school*, assigning it the modifier N/N rather than the noun N, even though both are provided by the supertagger. This has caused a number of incorrect rule choices in order to combine the categories together, though the remainder of the sentence is identical between the baseline and gold standard.

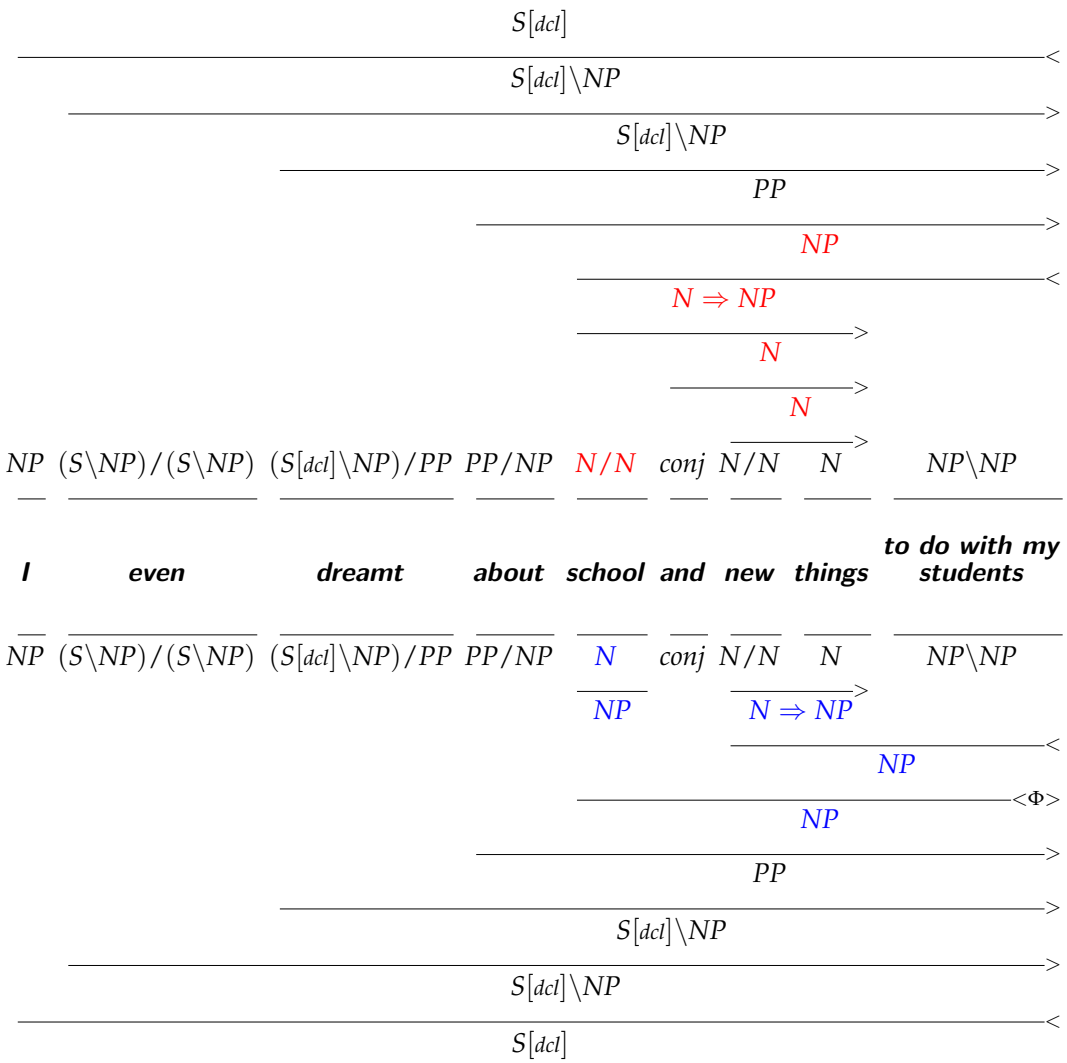
Intuition suggests that forcing the parser to construct the desired coordination between *school* and *things* would repair the sentence and produce the correct dependencies. The constraints required for this are:

$$\langle \textit{and}, \textit{conj}, 1, \textit{school} \rangle$$

$$\langle \textit{and}, \textit{conj}, 1, \textit{thing} \rangle$$

Strict enforcement of this correct coordination allows the parser to perfectly reproduce the gold standard. Applying these constraints also produces the correct parse under the remaining three constraint configurations:

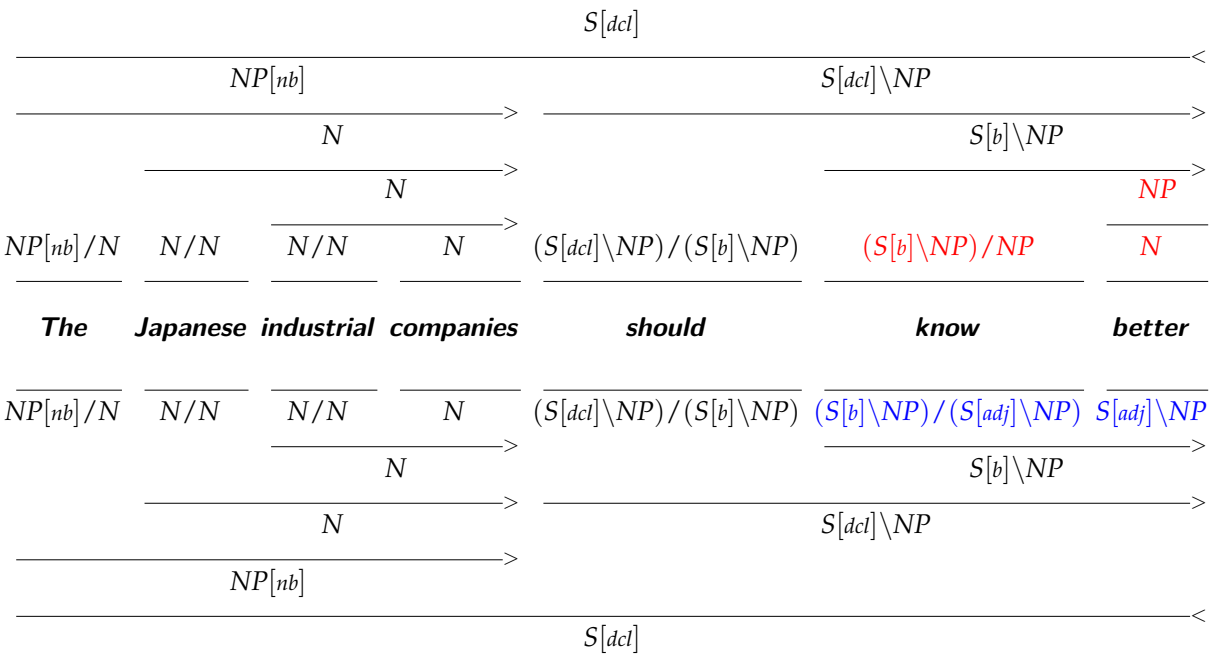
Add missing: the supertagger has provided all necessary categories, so this performs identically to strict enforcement.



Parse 7.3: The C&C analysis (top) and gold derivation (bottom) of CCGbank section 00 sentence. The parser has chosen an incorrect category for *school*, and chosen to apply the *conj* absorption rule rather than use the coordination combinator. Otherwise, the parses are identical.

Any category or slot: the only way for the parser to satisfy both head-filler constraints is to use the coordination rule and produce the *conj* dependencies. There is no other rule which would allow a dependency $\langle$*and*, *school*$\rangle$ simultaneously with $\langle$*and*, *things*$\rangle$ given the categories provided by the supertagger.

Constraints at root: there is already a parse which satisfies all constraints, so this performs identically to strict enforcement.



Parse 7.4: The C&C analysis (top) and gold derivation (bottom) of CCGbank section 00 sentence. The parser has chosen incorrect categories for *know* and *better* due to the supertagger not providing the desired category for the former. Otherwise, the parses are identical.

Parse 7.4 gives the C&C analysis and gold standard parse of another CCGbank section 00 sentence. Once again, the parser has chosen incorrect categories, with both *know* and *better* being incorrect. Intuitively, enforcing the following correct dependency between these words should allow the parser to match the gold standard:

$$\langle \textit{know},\ (S[b]\backslash NP)/(S[adj]\backslash NP),\ 2,\ \textit{better},\ - \rangle$$

Unfortunately, while the supertagger provided the $S[adj]\backslash NP$ for *better*, it has not provided the $(S[b]\backslash NP)/(S[adj]\backslash NP)$ category for *know*. The supertagger does actually not produce this category with any β value used by the parser, and requires even more permissive settings to return it. It is impossible for the parser to satisfy the constraint given the missing category, so under strict enforcement without fallback, the sentence is discarded as unparseable. When using fallback, the baseline analysis is returned with the erroneous categories and dependency. The other configurations react as thus:

Add missing: the required category is added to the chart, and the parser successfully satisfies the constraint and returns the desired parse.

Any category or slot: the baseline parse fulfills the requirement of a dependency between *know* and *better*, and so the parser returns it.

Constraints at root: all parses in the chart violate the provided constraint; given the categories provided by the supertagger, the variable co-indexed with *better* becomes inactive as soon as it forms any dependency. This is the trigger condition for detecting a violated constraint (see Section 6.2), so the sentence is rejected as unparseable without fallback, and the baseline parse is returned when using fallback.

7.3 Results

In this section, we describe the results of applying the constraints of each error class to the C&C parser under various constraint configurations.

7.3.1 Applying all constraints

Table 7.1 lists the results of applying all of our constraints to the C&C parser on automatic POS tags. Each table gives the *coverage* over section 00, the total number of *effective* constraints which were not present in the baseline, but are present in the constrained parse, and the *percentage* of effective constraints as a proportion of the total number applied. As with dependency parsers, the effective percentage can be viewed as the class *error rate*. *Missing* constraints are not present when the constrained parse is evaluated. They occur for three reasons:

- the parser has produced the required dependency, but has assigned it a flag indicating that it should be ignored during evaluation for compatibility with CCGbank;
- the parser is unable to find a parse which strictly satisfies the constraints, but a relaxation allows the best non-satisfying parse to be produced instead. This

All attachments	cover	eff	miss	eff %	LF	ΔLF
Baseline	99.06	-	-	-	84.91	-
Reject on failure						
Strict enforcement	63.36	2459	11	5.41	99.76	10.49
Add missing	74.54	3623	16	7.97	99.70	12.52
Any category or slot	68.69	2490	503	5.48	97.72	9.33
Constraints at root	74.44	2947	436	6.48	98.74	9.80
Reparse on failure						
Strict enforcement	99.06	2459	3699	5.41	90.94	6.03
Add missing	99.22	3623	2579	7.97	93.64	8.73
Any category or slot	99.06	2490	3730	5.48	90.82	5.91
Constraints at root	99.06	2947	3409	6.48	91.85	6.94

Table 7.1: Coverage, number of constraints which corrected an error or were missing in the parser output, correction percentage, overall labeled F-score, and F-score improvement over the baseline for all constraints to the C&C parser on automatic POS tags over CCGbank 00. ΔLF is calculated over the constrained coverage.

occurs when any category or slot is permitted when satisfying constraints, or when unsatisfied constraints are permitted;

- the parser is running in the fallback configuration, and has reverted to the baseline parse. The missing constraints are those which are incorrect in the baseline.

Finally, the labeled F-score and F-score Δs to the baseline parser over the sentences covered by both the constrained and baseline systems are listed.

Our results are divided into two main groups, depending on whether the parser rejects sentences for which it cannot satisfy the provided constraints, or whether it falls back to the baseline parser without constraints. The former gives a true representation of the impact of constraints on coverage, while the latter gives a comparable F-score to the baseline. Within these groups, we examine the impact of enforcing constraints, adding any missing categories required for constraints to the chart, relaxing the strictness of constraint satisfaction, and allowing the parser to produce a parse with unsatisfied (but not violated) constraints.

7.3.1.1 Reject on failure

Strict Enforcement

Applying all constraints without a fallback mechanism has a severe impact on the parser's coverage. The achieved coverage is under 64%, comparing unfavourably against the dependency parsers in Chapter 3, where the lowest recorded coverage was over 93% on newswire. F-score under strict enforcement is 99.76%, highlighting the impact of the compromises made by Clark and Curran (2007b) in the C&C parser design. The limited category set used by the supertagger and parser, inconsistencies in dependency annotation from CCGbank, and the omission of rare rules all contribute to the reduced coverage and inability to score 100%, underscoring the challenge of efficiently parsing CCG without artificial restrictions.

The missing 11 constraints under strict enforcement occur when the parser has correctly produced the required dependency, but has assigned it a flag indicating that it should be ignored during evaluation for compatibility with CCGbank.

Add missing

Adding the missing categories required by constraints improves coverage to 74.54%, whilst maintaining an F-score of 99.70%. Over 11% of the uncovered sentences under strict enforcement are due to supertagger errors.

The parser cannot achieve 100% coverage even when adding missing categories to the chart for several reasons. There are parses which require a category or rule not implemented by the parser, as well as inconsistencies in CCGbank which will be discussed in Section 7.4. Additionally, the supertagger assigns some incorrect categories to words which do not head any constraints, but are merely consumed by other categories as fillers. As these words do not head any constraints, they will not have their correct category added in our procedure.

Any category or slot

Relaxing the category and slot requirement results in 68.69% coverage and an F-score of 97.67%. As expected, there are substantially more missing constraints under this configuration. Interestingly, effective constraint percentages remain at roughly the same levels as strict enforcement. The relatively small drop in F-score and similar constraint effectiveness, despite the large reduction in specificity, are optimistic results, similar to the relatively small coverage drop for ZPar in Section 3.8.1. The presence of so many constraints helps to reduce the possible options for the parser, even though it is only being guided by word-word constraints. What remains to be seen is how well the parser will perform with smaller numbers of word-word constraints.

Constraints at root

Allowing parses to be accepted if they have unsatisfied constraints has the best coverage of the four configurations, at 74.44%. Effective constraint percentage and F-score sit roughly between that of strict enforcement and add missing, at 6.48% and 91.85% respectively. As our implementation will prefer parses with more satisfied constraints, the coverage improvement represents sentences where there is a dependency in CCGbank which the parser is unable to create due to the limitations of the grammar implementation or supertagger errors. Detailed examples are discussed in Section 7.4.

When not enforcing categories and slots, the missing constraints must all map to some head-filler combination in the constraints for a sentence, even if the category and slot do not match. Here, the missing constraints are constraints which are completely unsatisfiable, but not violated by variable checking in the sentence.

7.3.1.2 Reparse on failure

Falling back to the baseline parser when constraints cannot be satisfied avoids the large coverage reduction of the previous setup. At the same time, it allows the parser to use the constraints only when it can, at the cost of reducing the overall F-score

improvement. Under this scenario, the missing constraints statistic includes situations when the constrained parser is unable to form an analysis, and falls back to the baseline.

Strict enforcement

Using fallback restores coverage under all configurations to match the baseline, though there are some instances where constraints allow the parser to analyse sentences which the baseline previously could not, leading to slightly increased coverage. Strictly enforcing constraints with fallback results in an F-score of 90.94%, which is only 0.5% less than the 50-best oracle score with dependency hashing in Table 5.9. The difference between the two is due to situations where applying constraints fails, and the baseline parse is returned. The n-best parser can still return an analysis which is slightly more accurate, even though a perfect parse is not possible. There is no change in the effective constraint percentage in these experiments, as the only difference is that previously uncovered sentences are reparsed without constraints.

Add missing categories

Adding missing categories lifts coverage higher than the baseline parser, reaching 99.22% with an F-score of 93.64%. The combination of the constraints and the missing categories has allowed the parser to analyse sentences which the baseline configuration could not. Adding the missing categories also improves F-score beyond the oracle score of 50-best parsing, as the latter is restricted by supertagging accuracy.

Any category or slot

The 90.82 F-score under this configurations is practically indistinguishable from strict enforcement, with identical coverage. Given the parser implementation, knowing the full set of head-filler word-word pairs for a sentence is as beneficial as knowing the required slot and categories for those pairs. This demonstrates how restricted the

NP attachment	cover	eff	miss	eff %	LF	ΔLF	Δc	Δu
Baseline	99.06	-	-	-	84.91	-	-	-
Strict enforcement	99.06	1686	1206	12.25	91.36	6.45	4.30	2.17
Add missing	99.32	2174	760	15.79	93.32	8.41	5.40	2.93
Any category or slot	99.06	662	2372	4.81	86.51	1.60	1.45	.15
Constraints at root	99.06	1755	1187	12.75	91.36	6.45	4.48	1.98

Table 7.2: Coverage, number of constraints which corrected an error or were missing in the parser output, correction percentage, overall labeled F-score, and F-score improvement over the baseline for NP attachment constraints to the C&C parser on automatic POS tags over CCGbank 00. Δs are calculated over the constrained coverage.

parser is when applying so many constraints: despite the additional leeway in slot and category, it is very difficult to not assign the desired parse.

Whilst our results with fallback are substantially below the near-perfect F-scores achieved without fallback, they are much more easily comparable to the baseline parser and each other. The remainder of this section will compare the performance solely using the fallback scheme. We will also use the partitioning procedure described in Section 7.1 to determine the constrained and cascaded dependency impact.

7.3.2 NP attachment constraints

NP attachment constraints are the largest error class, and they repair the most parser errors. This is consistent with the results for dependency parsers in Section 3.8.1. Applying NP attachment constraints improves F-score by 6.45% over the baseline; missing categories contribute to nearly 2% additional F-score on top of this. The effective constraint percentage is double the overall percentage, at 12.25% under strict enforcement and 15.79% when adding missing categories. The C&C parser makes substantial NP errors.

The 6.45% ΔLF is divided into 4.30% Δc and 2.17% Δu, and this trend carries over each configuration. This is not unexpected in CCG; the assumed right-branching

NP internal	cover	eff	miss	eff %	LF	ΔLF	Δc	Δu
Baseline	99.06	-	-	-	84.91	-	-	-
Strict enforcement	99.06	340	156	3.38	86.19	1.28	.88	.40
Add missing	99.06	413	83	4.10	86.43	1.52	1.08	.44
Any category or slot	99.06	273	228	2.71	85.80	.89	.68	.21
Constraints at root	99.06	316	180	3.14	86.08	1.17	.80	.36

Table 7.3: Coverage, number of constraints which corrected an error or were missing in the parser output, correction percentage, overall labeled F-score, and F-score improvement over the baseline for NP internal constraints to the C&C parser on automatic POS tags over CCGbank 00. Δs are calculated over the constrained coverage.

structure of NPs makes identifying noun heads relatively simple, and NPs themselves are rarely the heads of non-NP dependencies, restricting the cascading impact. This is in contrast to dependency parsing, where incorrect NP attachments were responsible for near equal numbers of other errors. While NPs are a large source of error for both types of parsers, they cause more constrained impact for C&C, and more cascaded impact for dependency parsers.

Reducing the number of applied constraints also severely reduces the effectiveness of not enforcing categories and slots. ΔLF is just 1.60%, effective percentage drops substantially to 4.81%, and Δu is negligible at 0.15%. This illustrates how optimistic the results in this configuration were when applying all constraints, and this drop in comparative performance is consistent over all of our error classes.

7.3.3 NP internal constraints

NP internal constraints are the second most frequent error class. However, all NPs in CCGbank are assumed to be right-branching (Hockenmaier and Steedman, 2007), making this error class relatively easy to parse. The effective constraint percentage is below 4.5% across all configurations, while no configuration achieves more than 1.52 ΔLF. This is consistent with the dependency parsing results in Section 3.8.1, where

Modifier attachment	cover	eff	miss	eff %	LF	ΔLF	Δc	Δu
Baseline	99.06	-	-	-	84.91	-	-	-
Strict enforcement	99.06	466	243	8.73	86.86	1.95	1.19	.76
Add missing	99.06	579	130	10.84	87.42	2.51	1.49	1.02
Any category or slot	99.06	351	365	6.57	86.15	1.24	.89	.35
Constraints at root	99.06	453	259	8.48	86.73	1.82	1.14	.68

Table 7.4: Coverage, number of constraints which corrected an error or were missing in the parser output, correction percentage, overall labeled F-score, and F-score improvement over the baseline for Modifier attachment constraints to the C&C parser on automatic POS tags over CCGbank 00. Δs are calculated over the constrained coverage.

NP internal constraints produced one of the smallest overall accuracy improvements. While CCG NPs are much easier to parse than the Stanford NPs, disambiguating NP internal structure remains a strength across both.

Allowing constraints at the root no longer outperforms strict enforcement as it has in the larger error classes. With fewer constraints, there are fewer parses rejected by the stricter configurations, reducing the coverage advantage of allowing constraints at root and negating the resulting F-score advantage. This observation holds for all of the remaining error classes.

As with NP attachment constraints, the F-score improvement is skewed towards Δc, though some of the cascaded impact deltas are almost inconsequential.

7.3.4 Modifier attachment constraints

Modifier attachment errors occur much more frequently than NP internal errors, despite being a much smaller class. The contribution of constrained and cascaded dependencies for these constraints is less skewed than NP constraints across all configurations. For example, the 1.95% ΔLF under strict enforcement is split into 1.19% Δc and 0.76% Δu. In CCG, distinguishing modifiers from arguments, which are subcategorised for, can have substantial cascading impacts (as illustrated in Section 6.1).

Clause attachment	cover	eff	miss	eff %	LF	ΔLF	Δc	Δu
Baseline	99.06	-	-	-	84.91	-	-	-
Strict enforcement	99.06	324	242	8.88	86.56	1.65	.77	.87
Add missing	99.06	437	129	11.97	87.31	2.40	1.06	1.34
Any category or slot	99.06	243	331	6.66	85.93	1.02	.60	.41
Constraints at root	99.06	318	254	8.71	86.52	1.61	.76	.85

Table 7.5: Coverage, number of constraints which corrected an error or were missing in the parser output, correction percentage, overall labeled F-score, and F-score improvement over the baseline for Clause attachment constraints to the C&C parser on automatic POS tags over CCGbank 00. Δs are calculated over the constrained coverage.

However, knowing the endpoints and directionality of the expected dependency is often enough to disambiguate modifier attachments, as shown by the 1.24% ΔLF when allowing any category or slot. This is substantially higher than NP internal attachments and much closer to the strict enforcement improvement, suggesting that modifiers typically have only one category compatible with each attachment point.

7.3.5 Clause attachment constraints

Enforcing Clause attachment constraints shows a higher error rate than modifiers, but leads to smaller ΔLF improvements as it is a smaller error class. The improvements are skewed towards cascaded impact for the stricter configurations, implying that the clause errors made by C&C have a more substantial impact on the remainder of the sentence than the magnitude of the errors themselves. However, when allowing any categories or slots, the cascaded improvement is dramatically reduced relative to the constrained improvement. The effective constraint percentage in this configuration is 6.66% — only 2.22% less than strict enforcement — but the F-score improvement over cascaded dependencies has halved; 0.41% compared to 0.87%. This could be because in CCG, clause attachments typically involve two complex categories. Choosing the wrong

PP attachment	cover	eff	miss	eff %	LF	ΔLF	Δc	Δu
Baseline	99.06	-	-	-	84.91	-	-	-
Strict enforcement	99.06	881	271	18.62	90.03	5.12	2.01	3.11
Add missing	99.06	1018	134	21.51	91.07	6.16	2.34	3.82
Any category or slot	99.06	787	375	16.63	89.05	4.14	1.71	2.43
Constraints at root	99.06	869	289	18.36	89.89	4.98	1.96	3.03

Table 7.6: Coverage, number of constraints which corrected an error or were missing in the parser output, correction percentage, overall labeled F-score, and F-score improvement over the baseline for PP attachment constraints to the C&C parser on automatic POS tags over CCGbank 00. Δs are calculated over the constrained coverage.

category for either side of a head-filler constraint prevents other correct dependencies from being formed.

7.3.6 PP attachment constraints

PP attachment was challenging for the dependency parsers, and they remain so for C&C. These constraints have the highest effective percentages of all error classes, and their F-score improvement over the baseline is second only to NP attachment, which has three times the number of constraints. However, for C&C, PPs have a far greater cascading impact than they did for dependency parsers in Section 3.8.1. They are the only error class for CCG where Δu exceeds Δc, and this illustrates how the presence of head categories in CCG dependencies dramatically affects the evaluation, Prepositions almost always head dependencies (usually NP complements), so when an incorrect attachment point changes the category on the preposition, the complement dependency must also be corrected.

7.3.7 Coordination attachment constraints

Coordination dependencies are usually ignored during CCG English parsing evaluation as CCGbank does not include *conj* dependencies. To evaluate our Coordination

Coordination attachment	cover	eff	miss	eff %	LF	ΔLF	Δc	Δu
Baseline	99.06	-	-	-	83.80	-	-	-
Reparse on failure								
Strict enforcement	99.06	336	593	8.88	84.76	.96	.81	.15
Add missing	99.06	444	485	11.74	84.80	1.00	1.02	-.02
Any category or slot	99.06	348	581	9.20	84.78	.98	.85	.13
Constraints at root	99.06	284	649	7.51	84.44	.64	.73	-.08

Table 7.7: Coverage, number of constraints which corrected an error or were missing in the parser output, correction percentage, overall labeled F-score, and F-score improvement over the baseline for Coordination attachment constraints to the C&C parser on automatic POS tags over CCGbank 00. Evaluation includes *conj* dependencies. Δs are calculated over the constrained coverage.

attachment constraints, we add the extracted *conj* dependencies from Section 6.3.2 to CCGbank, and prevent them from being ignored in the evaluation script. These results are not directly comparable with the standard section 00 evaluation elsewhere in this chapter, but they allow us to accurately quantify the effect of the constraints.

Coordination is frequently incorrect in the parser, with a higher effective percentage than NP internal, Modifier attachment, or Clause attachment. However, it has a smaller ΔLF than all of those error classes, even though the parser is starting from a lower baseline. The bulk of ΔLF comes from the constrained impact; enforcing correct coordination provides almost no cascaded improvement under any configuration. This suggests that the constraints introduce as many errors as they repair in the parser.

Coordination attachment is the only error class to exhibit this behaviour, and it is explained by the nature of coordination in CCG. Every time a coordination occurs, it duplicates every dependency that the coordinated items appear in, and forces those items to share the same category. However, unlike the other error classes, the constraints do nothing to ensure that the coordinated items have the correct category — they merely ensure that the *conj* and its filler words are correct. The parser is free to choose any categories provided by the supertagger which are compatible with the coordination

Root attachment	cover	LF	ΔLF
Baseline	99.06	84.91	-
Strict enforcement	99.06	85.96	1.05
Add missing	99.11	86.31	1.40
Any category or slot	99.06	85.08	.17

Table 7.8: Coverage, overall labeled F-score, and F-score improvement over the baseline for Root attachment constraints to the C&C parser on automatic POS tags over CCGbank 00. Δs are calculated over the constrained coverage.

— whether these are correct for the rest of the parse or not. This is in contrast to dependency parsers, where coordination constraints never result in a negative cascaded impact. We will illustrate this further in Section 7.4.

7.3.8 Root attachment constraints

Applying Root attachment constraints requires a different procedure to all other error classes, so it is not possible to calculate an effective percentage or partition the resulting dependencies. It is also not meaningful to allow constraints at the root of the tree because only one constraint is being applied, and it should be on the root.

Strictly enforcing the category and position of the root word improves F-score by 1.05% over the baseline to 85.96%. Adding the category for the root word if it is missing from the chart increases these to 1.40% and 86.31% respectively. These results are larger F-score improvements than applying NP internal constraints, even though the latter class is much larger.

Allowing the root to have any category dramatically reduces the F-score improvement to just 0.17% over the baseline. In this configuration, the parser is allowed to choose the highest scoring category at the top of the tree as long as the desired root word is the active X variable on that category. When the parser does choose an erroneous root, it is very often carrying an incorrect category, which has cascading effects on the overall F-score.

7.4 Constraints for Parser Debugging

Developing a CCG parser is a complex task, particularly due to the non-standard rules introduced in CCGbank to handle the inconsistencies and under-specifications inherent in the Penn Treebank annotation. We have shown how our CCG constraint procedure can serve as an evaluation tool for the parser, illustrating performance and cascading impact across meaningful error classes. However, our procedure can also serve as a robust method for uncovering deficiencies in the parser which prevent C&C from constructing certain expected dependencies.

In this section, we present a number of sentences from CCGbank section 00 which failed to be parsed when strictly enforcing constraints. Each parse fails for a different reason, highlighting different problems.

7.4.1 Supertagger Errors

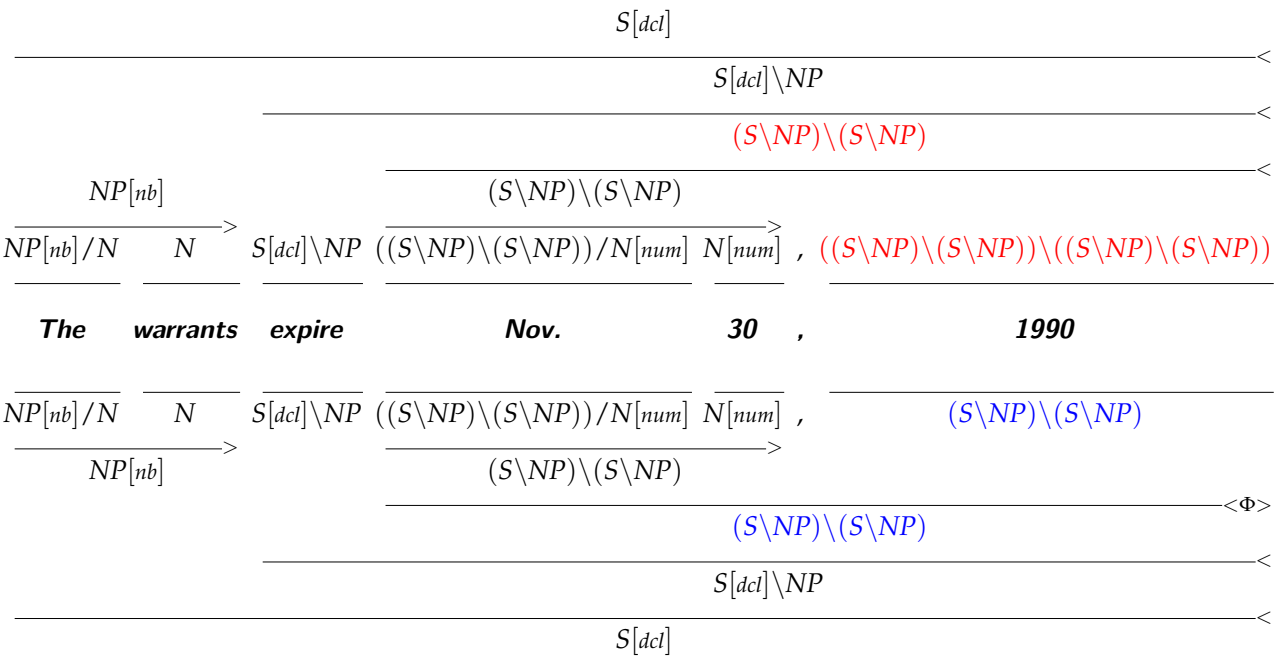
If the supertagger does not assign the correct category to a word, it is impossible to satisfy any constraints headed by that word. When this happens, the parser will fail to find an analysis for the sentence. The difference in coverage between strict enforcement and adding missing categories illustrates how often this occurs.

The supertagger uses POS tags as core features, and its accuracy varies from 97.34% to 99.17% on gold POS tags using the default β levels in the C&C parser, while automatic POS tags reduce this range to 96.34% to 98.66% (Clark and Curran, 2007b).

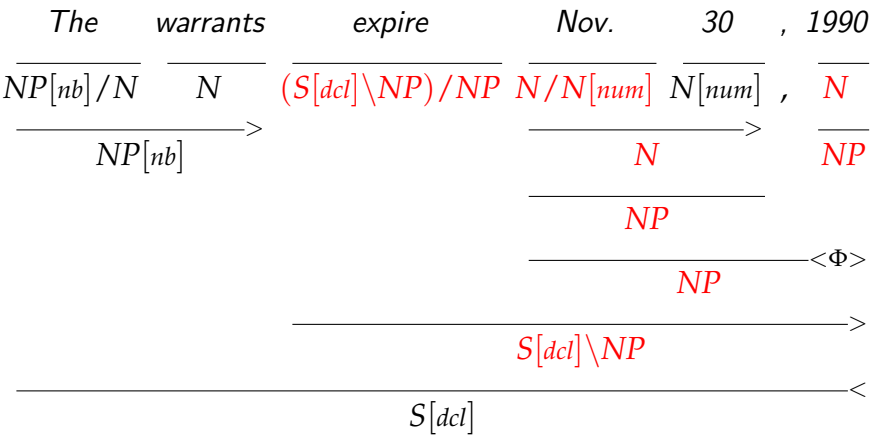
Parse 7.5 depicts a CCGbank section 00 sentence, and the analysis returned by the baseline C&C parser. The parser has made one category error on *1990*, but the remaining categories are correct. The labelled F-score of this sentence is 86.3%, as there are four correct, one missing, and one incorrect dependencies.[1]

Parse 7.6 depicts the parser output when we enforce the correct coordination of *Nov.* and *1990* using *conj* constraints. Unfortunately, the supertagger does not provide

[1] *conj* dependencies are ignored in the standard evaluation.

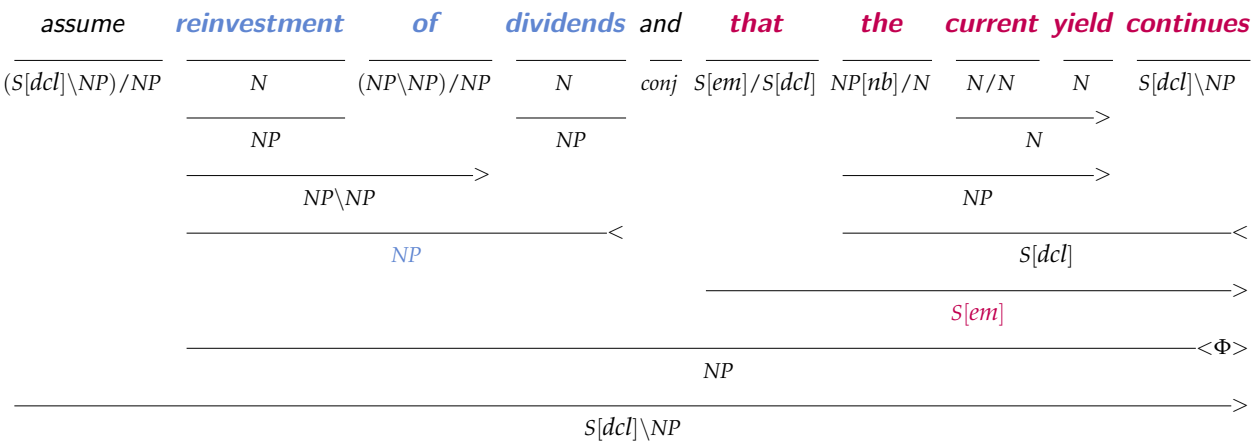


Parse 7.5: The C&C analysis (top) and gold parse of a CCGbank section 00 sentence. The parser has chosen an incorrect category for *1990*, which should be coordinated with *Nov.*



Parse 7.6: The C&C analysis when enforcing the coordination of *Nov.* and *1990*. The supertagger does not return the necessary $(S\backslash NP)\backslash(S\backslash NP)$ category for *1990*, but the parser has managed to use the categories provided to satisfy the coordination constraints. Unfortunately, the F-score is reduced due to the changes in category for *expire* and *Nov.*

the parser with the correct category for *1990*, so it cannot construct the gold parse. However, the parser has still managed to satisfy the constraints, but at the cost of reducing F-score to just 10%, as there is now only one correct evaluated dependency



Parse 7.7: Non-equivalent coordination in CCGbank. The categories *NP* and *S*[*em*] are coordinated to form an *NP*; this is impossible with any standard CCG rule.

(between *The* and *warrants*), four incorrect, and four missing dependencies. This example highlights the counter-intuitive impact of applying constraints to CCG, and how constraints can actually reduce F-score dramatically depending on the actual source of error in a parse. It also illustrates why Coordination attachment constraints have such poor cascading F-score: by only specifying the coordination, and not the structure of the coordinated elements, the parser often makes errors elsewhere in the parse.

7.4.2 Non-standard CCG Rules in CCGbank

There are a number of non-standard rules implemented in the CCGbank corpus to handle difficult or inconsistently annotated constructions in the Penn Treebank. Only a subset of these rules are implemented in the C&C parser as they can introduce inconsistencies or large inefficiencies. One example is *unlike coordination*, where two unlike constituents are coordinated together. While CCG elegantly handles regular coordination, it has no mechanism for coordinating two constituents with different categories. Parse 7.7 shows an example of how CCGbank handles coordinating an *NP* and $S[em]$ together to form the subject for a verb.

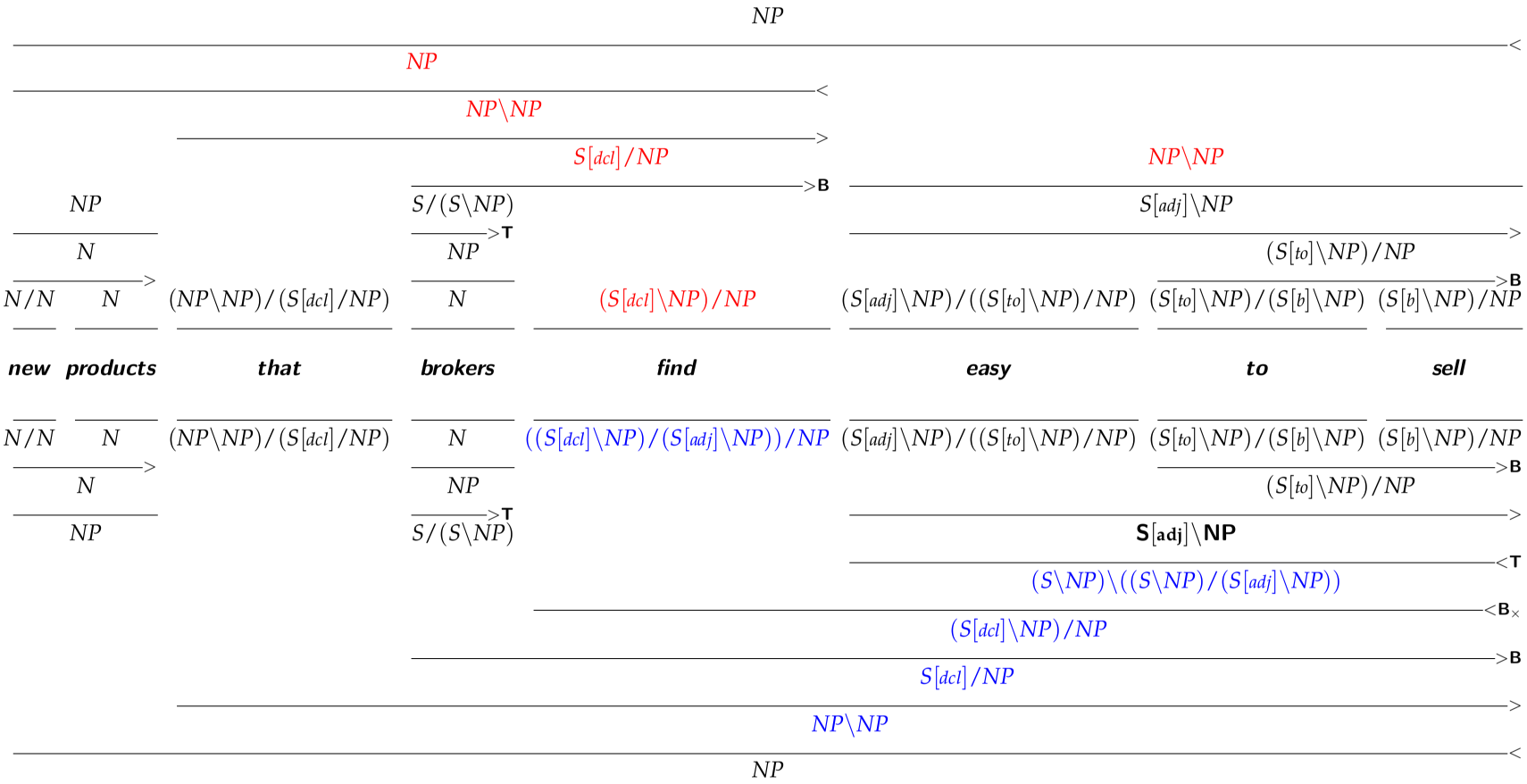
This type of coordination is rare, so this rule is not implemented in the C&C parser to prevent overgeneration. As a result, the parser cannot find the desired analysis even when the supertagger provides the correct categories. Dependencies mediated across this coordination cannot be enforced with constraints.

7.4.3 Inconsistencies between dependencies and the derivation

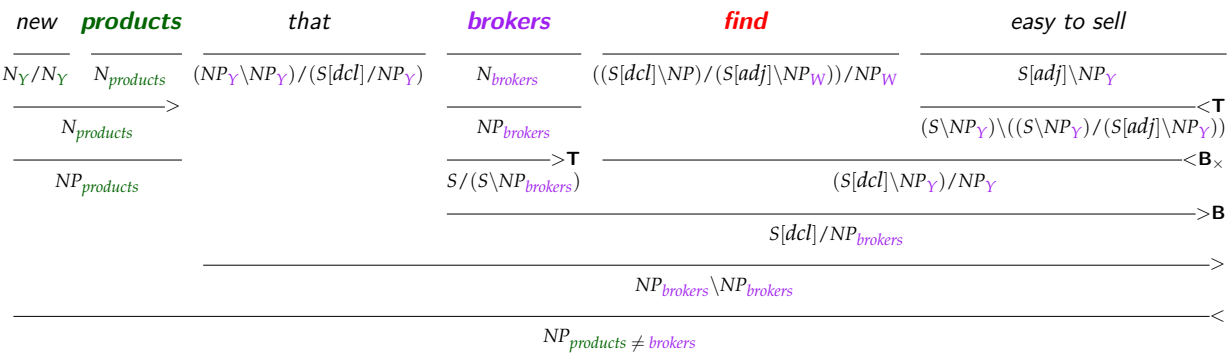
There are further cases where the CCG dependencies specified by CCGbank are inconsistent with the provided derivation. In Parse 7.8, the object of *sell* is *products*, so a long-range dependency is expected between the two. The fully annotated category of *sell* is $(S[b]\backslash NP^1_Y/NP^2_Z)$, so the object *products* maps to the Z variable in slot 2.

The crucial category is the bolded $\mathbf{S[adj]\backslash NP}$. This is used to analyse attributive adjectival phrases in copula constructions (Hockenmaier and Steedman, 2005). In the sentence *The computers were crude*, the copula *were* separates the subject *computers* from its adjective *crude*. The adjective requires a complex category with a $\backslash NP$ argument to carry a variable that will eventually unify with the subject to form a dependency. The $S[adj]$ result is a synthetic atomic category which is never created; it is merely present to complete the complex category required to carry the variable and unfilled dependency across the copula to the subject. The copula drives the interaction, consuming and removing the $S[adj]$ from *crude* before consuming *computers* and filling the dependency.

Once the gold-standard derivation in Parse 7.8 reaches the bolded $\mathbf{S[adj]\backslash NP}$, its fully annotated category $(S[adj]\backslash NP^1_Y)$ no longer has a Z variable, so the required dependency between *sell* and *products* can no longer be created. The adjectival phrase category can only carry a subject dependency, so the required object dependency for *sells* is impossible to satisfy.



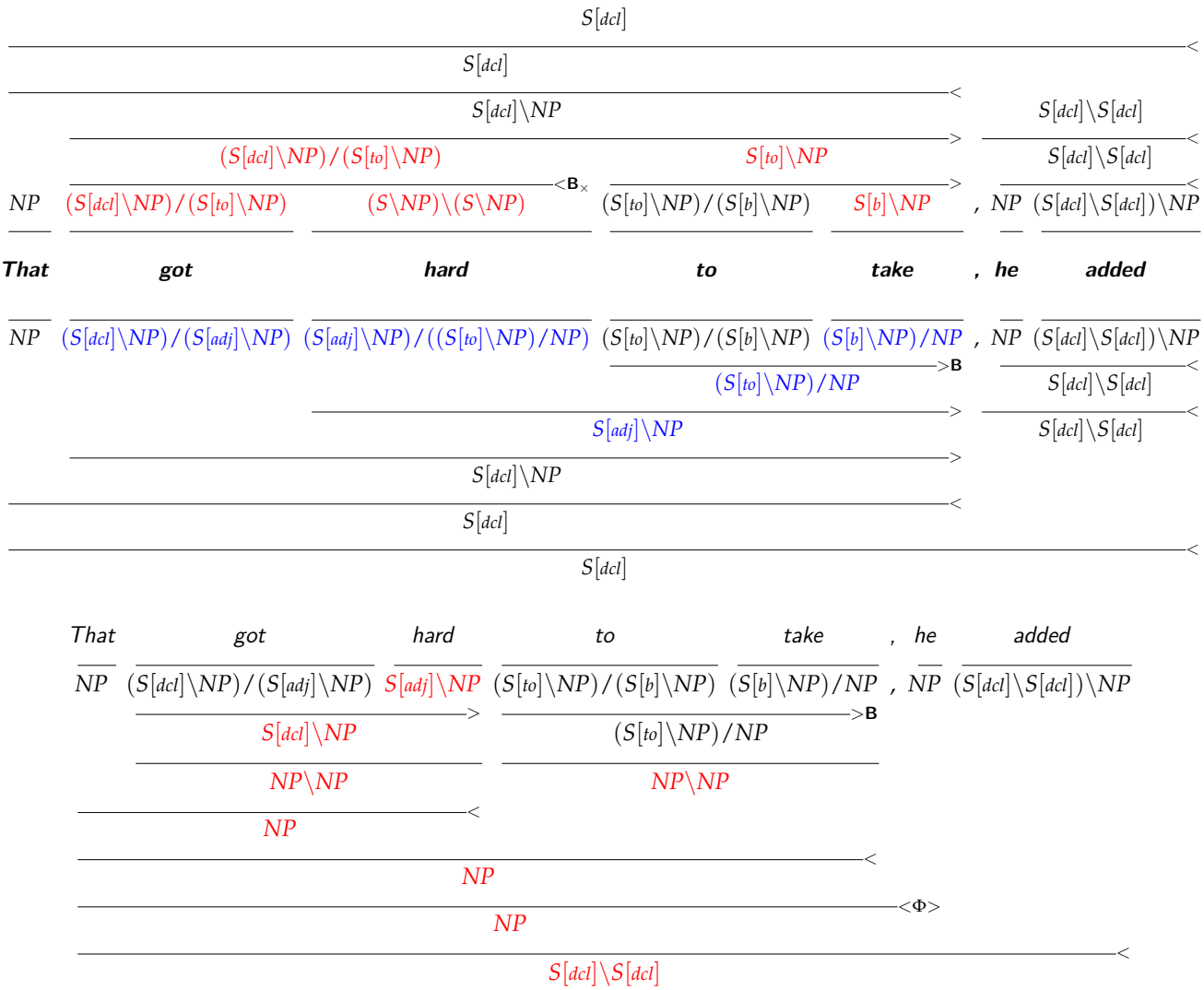
Parse 7.8: The C&C baseline (top) and gold parse (bottom) of a CCGbank section 00 sentence. A dependency is expected between *sell* and *products*, using variable Z and slot 2. However, at the bolded $\mathbf{S[adj]\backslash NP}$, there is no Z variable remaining, so the dependency cannot be formed.



Parse 7.9: The final backwards application is blocked, as the X variable cannot be unified. On the left, the NP is headed by *products*, but on the right, $NP\backslash NP$ expects an argument headed by *brokers*. This is ultimately due to the co-indexation for object control, rather than subject control on the category for *find*.

7.4.4 Co-indexation Inconsistencies in Categories

Setting aside the missing dependency, the supertagger provides the necessary categories to allow the parser match the gold standard, but no combination of constraints can induce the parser to create the gold parse in Parse 7.8. Hockenmaier and Steedman (2007) notes that the co-indexation in CCGbank is based on the assumption that words bearing the same CCG category always produce the same dependencies. This is insufficient for control verbs, which are divided into *subject* control verbs, and *object* control verbs. CCGbank co-indexes categories for object control only, but the gold-standard derivation requires subject control driven by *find*. The annotation on the gold-standard category is $((S[dcl]\backslash NP)/(S[adj]\backslash NP_W))/NP_W$, co-indexing the outermost NP argument with the VP object NP *brokers*. This ultimately blocks the desired gold-standard derivation from being constructed, as it requires the outermost NP argument to be co-indexed with the VP subject NP, *products*. Adjusting the co-indexation on *find* allows the gold-standard derivation to be formed, avoiding the rejected unification (though the dependency that is missing from the previous section remains missing). Parse 7.9 illustrates the values of each co-indexed variable in the derivation using the incorrect annotation, demonstrating how conflicting NP heads are ultimately set.



Parse 7.10: The C&C analysis (top), gold derivation (middle), and derivation with NP attachment constraints (bottom) of CCGbank section 00 sentence. The baseline parser cannot create the desired parse due to the co-indexation on $(S[dcl]\backslash NP)/(S[adj]\backslash NP)$. The parser mangles the parse in order to satisfy the constraints.

Co-indexation inconsistencies do not just prevent the parser from being able to find an analysis of a sentence. In some cases, partial constraints can force the parser to construct a worse derivation than the baseline, even though the constraints are satisfied. Parse 7.10 depicts a short sentence from CCGbank section 00, where the baseline parse has three incorrect categories, but still manages to construct a parse which is reasonably

similar to the gold standard. The incorrect categories have resulted in only one missing NP attachment from *take* to its object *That*; all other dependencies are correct.

When we enforce the correct NP attachment, the parser only chooses one incorrect category for the word *hard*. However, the wrong category is chosen because the co-indexation on the desired category does not allow the dependency between *That* and *take* to be formed. Instead, the parser manages to find a way to create the desired dependency using unary rules, producing a final parse with the sentential modifier $S[dcl]\backslash S[dcl]$ at the top rather than the expected $S[dcl]$. The constrained derivation has introduced one incorrect and four missing dependencies compared to the baseline, even though we have successfully retrieved the single missing dependency.

7.4.5 Categories Not Implemented in the Parser

While CCGbank contains over a thousand different lexical categories, the C&C supertagger and parser use a limited subset of 425 for efficiency. If a constraint specifies a head category that is not used in the parser, then the constraint is impossible to satisfy, and the parser will fail to find a spanning analysis for the sentence. This issue prevents the parser from achieving 100% F-score even using perfect categories, and is referred to as *grammar error*. Unfortunately, there is no way to address this problem aside from including the additional categories to the parser, potentially reducing its efficiency and affecting overall accuracy in the process.

7.5 Summary

In this chapter, we have described how to perform a constraint-based evaluation of the C&C parser.

We defined a procedure to partition the dependencies from a baseline and constrained CCG parse, producing constrained and cascaded dependency sets for evaluation. This allows us to determine the associated impacts on F-score, analogous to Δc

and Δu for dependency parsers in Chapter 3. Heuristically identifying corresponding dependencies which have changed between the baseline and constrained parse is an important part of this process.

The relative contribution of each error class is similar between C&C and the dependency parsers in Chapter 3. However, there are substantial differences in how the errors are distributed between the different parsers. Repairing NP and PP attachments gives rise to large increases in parser accuracy. However, NP constraints cause much more cascading impact in dependency parsers than C&C, while the reverse is true for PP constraints. This is largely a function of the strict CCG evaluation procedure. Coordination constraints have negligible or even negative cascaded impact for CCG, while this is not the case for dependency parsers.

Punctuation is another point of difference between the formalisms. In CCG, punctuation is typically absorbed without affecting the logical form of a sentence, while in dependency schemes punctuation is often attached to sentence roots and phrasal boundaries, affecting the projectivity applied over the sentence. Repairing erroneous punctuation creates substantial cascading impact in dependency parsers, illustrating how it serves as a marker for further errors. Correctly attaching punctuation clearly helps the parsers return the correct analysis. In CCG, punctuation cannot have the same effect.

Applying constraints has a severe impact on the coverage of C&C. Less than 65% of CCGbank section 00 is covered when strictly enforcing all constraints. 11% more sentences are covered by eliminating supertagger error. There are a number of reasons for the remaining coverage deficit. C&C only implements a subset of the rules and categories in CCGbank, meaning that some constraints cannot possibly be satisfied. Additionally, there are inconsistencies in derivations and category annotations from CCGbank which prevent the parser from successfully building some desired analyses. Constraints have allowed us to quantify the impact of the supertagger on parsing

accuracy, and examine the how the compromises made by Clark and Curran (2007b) prevent the parser from achieving perfect accuracy.

We have demonstrated the intricacies of applying our constraint-based evaluation procedure in CCG, and how it is useful for investigating parser performance and implementation. We have contrasted our findings with our work in Chapter 3 on dependency parsers. In the next chapter, we will summarise our main findings, reflect on the issues that we have raised, and discuss future work.

8 Conclusion

Identifying and addressing the remaining errors in parsing is important for improving downstream applications in natural language processing. The core contribution of this thesis is a constraint-based procedure to quantify the errors made by parsers, and identify potential causes and consequences of errors. A strength of our approach is the ability to separate the direct constrained impact of corrections from constraints from the indirect cascading impact of the parser changing its analysis in response to constraints. This avoids assumptions regarding the at times unpredictable behaviour of parsers, while quantitatively illustrating the effects of parser errors.

In Chapter 3, we implemented constraint-based evaluation for the graph-based MSTParser and transition-based ZPar dependency parsers, comparing their performance on newswire and web text. Chapters 6 and 7 described the constraint evaluation implementation for the C&C Combinatory Categorial Grammar parser. ZPar was more accurate than MSTParser overall, but also exhibited more cascading improvement when constraints were applied, despite requiring fewer corrections. The transition-based model was better able to translate enforced arcs into wider improvements, despite, or perhaps due to, pruning a large proportion of its search space through beam search.

We found that NP and PP attachments are particular challenges across all parsers, but affect parsing accuracy in different ways. PPs are localised errors for dependency parsers, exerting limited influence when wrongly attached. However, they cause substantial cascading impact for CCG due to the change in head category that often comes with a change in attachment, and the harsh evaluation metric for CCG parsers. The

converse is true for NP attachments, where cascading impact is as substantial as constrained impact for dependency parsers, but less so for CCG. For dependency parsers, punctuation errors are also a large source of cascading errors; incorrect punctuation is a strong indicator of an erroneous parse, despite being typically ignored in evaluation and inconsistently treated in treebanks. Most of our other error classes behave consistently across the formalisms, parsers, and domains.

Features derived from *n*-gram frequency counts in unannotated corpora have been shown to assist with NP and PP attachments, and we experiment with surface and syntactic *n*-gram features for MSTParser in Chapter 4, building on the work of Bansal and Klein (2011). Both surface and syntactic *n*-gram features perform similarly in isolation across dependency schemes and domains, though syntactic *n*-grams work best out-of-domain. The feature types are also complementary, with a combined system of surface and syntactic *n*-gram features outperforming all other feature types, achieving up to 1.3% UAS improvements in-domain and 1.6% out-of-domain. We find that, while these features are successful in addressing errors such as NP and PP attachments, they do not fundamentally change the overall error distribution, and primarily serve to correct constrained errors rather than reduce cascaded impact.

CCG parsing is complex, as different derivations can generate identical dependencies. Algorithms which are straightforward for constituency and dependency parsers are not necessarily easy to implement for CCG. We show the intricacies of our constraint evaluation procedure for CCG, and describe how applying constraints also provides an evaluation of the implementation of the parser itself. Our experiments with *n*-best CCG parsing in Chapter 5 demonstrate another task where complications arise from derivations being semantically identical. We find that *n*-best parsing algorithms designed without CCG's dependency formulation in mind produce redundant *n*-best parses. We design dependency hashing as an efficient solution to this issue, and demonstrate how it improves the performance of a CCG reranker.

8.1 Future Work

Our constraint-based evaluation procedure allows dependency parsers to be compared across high-level error classes, illustrating how the cascading impact of constraints can be quite different for each system. Applying this work to other popular dependency parsers is an obvious extension; it will be interesting to see if other dependency parsing formulations present a different distribution of constrained and cascading errors across our classes. Defining new error classes across different dependency schemes will allow our procedure to be applied to a diverse range of corpora, parsers, and languages.

Another direction is to apply our constraint-based technique to constituency parsers. Rather than using dependencies, span-level constraints could be enforced whilst otherwise allowing the parser to choose the remainder of the analysis. This would provide a direct comparison against the work of Kummerfeld et al. (2012), and highlight the differences between our procedure and theirs.

There are few CCG parsers which directly generate CCG dependencies like C&C. Most recent work has produced parsers which create derivations using shift-reduce (Zhang and Clark, 2011a; Xu et al., 2014), sequence tagging (Lewis and Steedman, 2014b), or constituency-based techniques (Fowler and Penn, 2010), and then employ C&C's `generate` tool to convert the derivations to dependencies. This makes broadly applying our constraint-based CCG evaluating difficult. However, some recent work is based directly on C&C, e.g. Auli and Lopez (2011a,c), and it would be instructive to apply our constraint procedure to these systems to examine the effect of different parsing algorithms and optimisation procedures on error classes in CCG.

The complementary nature of surface and syntactic n-gram features extracted from web-scale corpora suggest that further individual gains may be possible with features of this design. Testing these features on other dependency parsers, and continuing our postponed investigation of these features for ZPar, will show whether they are effective across higher quality parsers, or if they are particularly effective for parsers compatible

with MSTParser. Further experiments with relative counts in place of absolute counts may also be useful, as Chen et al. (2013) found that relative counts performed better in their meta-features parser.

Hashing over CCG dependencies completely solved the issue of redundant CCG parses. However, CCG dependencies are a fine-grained logical form representation, and certain relationships, such as the subjects of intransitive, transitive, and ditransitive verbs, are all represented by different CCG dependencies despite exhibiting similar behaviour. Implementing hashing over the formalism-independent grammatical relations in the parser would allow these similarities to be captured whilst still promoting diversity in *n*-best parses.

Our experiments evaluating dependency parsers on web text did not include any specific features tailored to each domain. Addressing the quality of POS tagging, and incorporating some element of the Web Treebank or other in-domain text as training data would improve the baseline out-of-domain performance, and allow us to quantify the impact across error classes of improving the inputs to the parser.

8.2 Summary

This thesis describes a technique for evaluating parsers based on applying dependencies as constraints, without assumptions on parser behaviour. It also proposes dependency hashing for CCG, a robust technique for efficiently summarising the logical form in CCG derivations. We have comprehensively demonstrated the effectiveness of our evaluation procedure across different parsers, formalisms, and domains, and identified how different error classes behave differently in parsers. Some errors cause much more cascading impact than others, suggesting that they should be prioritised in further efforts to address parser performance.

We hope this work will provide insights into parser performance, and lead to better syntactic analysis for downstream applications in natural language processing.

Bibliography

Kazimierz Adjukiewicz. 1935. Die syntaktische Konnexität. *Studia Philosophica*, pages 1–27.

Yoav Artzi, Dipanjan Das, and Slav Petrov. 2014. Learning Compact Lexicons for CCG Semantic Parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP-14)*, pages 1273–1283.

Abhishek Arun and Frank Keller. 2005. Lexicalization in Crosslinguistic Probabilistic Parsing: The Case of French. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL-05)*, pages 306–313.

Michael Auli and Adam Lopez. 2011a. A Comparison of Loopy Belief Propagation and Dual Decomposition for Integrated CCG Supertagging and Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-11)*, pages 470–480.

Michael Auli and Adam Lopez. 2011b. Efficient CCG Parsing: A* versus Adaptive Supertagging. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-11)*, pages 1577–1585.

Michael Auli and Adam Lopez. 2011c. Training a Log-Linear Parser with Loss Functions via Softmax-Margin. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP-11)*, pages 333–343.

Srinivas Bangalore and Aravind K. Joshi. 1999. Supertagging: An Approach to Almost Parsing. *Computational Linguistics*, 25(2):237–265.

Mohit Bansal, Kevin Gimpel, and Karen Livescu. 2014. Tailoring continuous word representations for dependency parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 809–815.

Mohit Bansal and Dan Klein. 2011. Web-Scale Features for Full-Scale Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-11)*, pages 693–702.

Yehoshua Bar-Hillel. 1953. A Quasi-Arithmetical Notation for Syntactic Description. *Language*, 29(1):47–58.

Yehoshua Bar-Hillel, Chaim Gaifman, and Eliyahu Shamir. 1960. On Categorial and Phrase-structure Grammars. In *The Bulletin of the Research Council of Israel*, pages 1–16.

Eduard Bejček, Eva Hajičová, Jan Hajič, Pavlína Jínová, Václava Kettnerová, Veronika Kolářová, Marie Mikulová, Jiří Mírovský, Anna Nedoluzhko, Jarmila Panevová, Lucie Poláková, Magda Ševčíková, Jan Štěpánek, and Šárka Zikánová. 2013. Prague Dependency Treebank 3.0.

Ann Bies, Justin Mott, Colin Warner, and Seth Kulick. 2012. English Web Treebank. LDC Catalog LDC2012T13.

Daniel Bikel. 2002. Design of a Multi-lingual, Parallel-processing Statistical Parsing Engine. In *Proceedings of the Second International Conference on Human Language Technology Research*, pages 178–182.

Ezra W. Black, Steven P. Abney, Daniel P. Flickenger, Claudia Gdaniec, Ralph Grishman, Philip Harrison, Donald Hindle, Robert J. P. Ingria, Frederick Jelinek, Judith L. Klavans, Mark Y. Liberman, Mitchell P. Marcus, Salim Roukos, Beatrice Santorini,

and Tomek Strzalkowski. 1991. A Procedure for Quantitatively Comparing the Syntactic Coverage of English Grammars. In *Proceedings of the 4th DARPA Speech and Natural Language Workshop*, pages 306–311.

Ezra W. Black, Roger Garside, and Geoffrey N. Leech, editors. 1993. *Statistically-driven computer grammars of English: The IBM/Lancaster approach*. Number 8 in Language and Computers. Rodopi, Amsterdam.

Thorsten Brants and Alex Franz. 2006. Web 1T 5-gram version 1. LDC Catalog LDC2006T13.

Forrest Brennan. 2008. *k-best Parsing Algorithms for a Natural Language Parser*. Master's thesis, University of Oxford, Oxford, United Kingdom.

Ted Briscoe. 2006. An Introduction to Tag Sequence Grammars and the RASP System Parser. Technical Report 662, University of Cambridge, Cambridge, United Kingdom.

Ted Briscoe and John Carroll. 2006. Evaluating the Accuracy of an Unlexicalized Statistical Parser on the PARC DepBank. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 41–48.

Ted Briscoe, John Carroll, Jonathan Graham, and Ann Copestake. 2002. Relational Evaluation Schemes. In *Proceedings of the Beyond PARSEVAL Workshop at the 3rd International Conference on Language Resources and Evaluation (LREC-02)*, pages 4–8.

Ted Briscoe, John Carroll, and Rebecca Watson. 2006. The Second Release of the RASP System. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, pages 77–80.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X Shared Task on Multilingual Dependency Parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-06)*, pages 149–164.

Xavier Carreras. 2007. Experiments with a Higher-Order Projective Dependency Parser. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 957–961.

John Carroll, Ted Briscoe, and Antonio Sanfilippo. 1998. Parser Evaluation: A Survey and A New Proposal. In *Proceedings of the 1st International Conference on Language Resources and Evaluation (LREC-98)*, pages 447–454.

Daniel Cer, Marie-Catherine de Marneffe, Dan Jurafsky, and Chris Manning. 2010. Parsing to Stanford Dependencies: Trade-offs between Speed and Accuracy. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC-10)*, pages 1628–1632.

Eugene Charniak. 1997. Statistical Parsing with a Context-free Grammar and Word Statistics. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 598–603.

Eugene Charniak. 2000. A Maximum-Entropy-Inspired Parser. In *Proceedings of the 1st Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-00)*, pages 132–139.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL-05)*, pages 173–180.

Wenliang Chen, Jun'ichi Kazama, Kiyotaka Uchimoto, and Kentaro Torisawa. 2009. Improving Dependency Parsing with Subtrees from Auto-Parsed Data. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP-09)*, pages 570–579.

Wenliang Chen, Min Zhang, and Yue Zhang. 2013. Semi-Supervised Feature Transformation for Dependency Parsing. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP-13)*, pages 1303–1313.

Yuchang Cheng, Masayuki Asahara, and Yuji Matsumoto. 2005. Chinese deterministic dependency analyzer: Examining effects of global features and root node finder. In *In Proceedings of the Fourth SIGHAN Workshop on Chinese Language Processing*, pages 17–24.

Stephen Clark and James R. Curran. 2003. Log-Linear Models for Wide-Coverage CCG Parsing. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing (EMNLP-03)*, pages 97–104.

Stephen Clark and James R. Curran. 2004a. Parsing the WSJ Using CCG and Log-Linear Models. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 103–110.

Stephen Clark and James R. Curran. 2004b. The Importance of Supertagging for Wide-Coverage CCG Parsing. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING-04)*, pages 282–288.

Stephen Clark and James R. Curran. 2007a. Formalism-Independent Parser Evaluation with CCG and DepBank. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL-07)*, pages 248–255.

Stephen Clark and James R. Curran. 2007b. Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. *Computational Linguistics*, 33(4):493–552.

Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building Deep Dependency Structures using a Wide-Coverage CCG Parser. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL-02)*, pages 327–334.

John Cocke and Jacob T. Schwartz. 1970. *Programming languages and their compilers: preliminary notes*. New York University, New York City, New York, USA.

Michael Collins. 1996. A New Statistical Parser Based on Bigram Lexical Dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL-96)*, pages 184–191.

Michael Collins. 1997. Three Generative, Lexicalised Models for Statistical Parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics (ACL-97)*, pages 16–23.

Michael Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia, Pennsylvania, USA.

Michael Collins. 2000. Discriminative Reranking for Natural Language Parsing. In *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 175–182.

Michael Collins. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP-02)*, pages 1–8.

Michael Collins, Jan Hajic, Lance Ramshaw, and Christoph Tillmann. 1999. A Statistical Parser for Czech. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL-99)*, pages 505–512.

Brooke Cowan and Michael Collins. 2005. Morphology and Reranking for the Statistical Parsing of Spanish. In *Proceedings of the 2005 Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP-05)*, pages 795–802.

James R. Curran, Stephen Clark, and David Vadas. 2006. Multi-Tagging for Lexicalized-Grammar Parsing. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING/ACL-06)*, pages 697–704.

Ido Dagan, Bill Dolan, Bernardo Magnini, and Dan Roth. 2009. Recognizing textual entailment: Rational, evaluation and approaches. *Natural Language Engineering*, 15(4):i–xvii.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-06)*.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008a. Stanford Dependencies manual. Technical report, Stanford University, Stanford, California, USA.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008b. The Stanford typed dependencies representation. In *COLING 2008 Workshop on Cross-framework and Cross-domain Parser Evaluation*, pages 1–8.

Bojan Djordjevic. 2006. Efficient Combinatory Categorial Grammar Parsing. In *Proceedings of the 2006 Australasian Language Technology Workshop (ALTW-06)*, pages 3–10.

Amit Dubey and Frank Keller. 2003. Probabilistic Parsing for German Using Sister-Head Dependencies. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL-03)*, pages 96–103.

Jason Eisner. 1996a. An Empirical Comparison of Probability Models for Dependency Grammar. Technical report, University of Pennsylvania.

Jason Eisner. 1996b. Efficient Normal-Form Parsing for Combinatory Categorial Grammar. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL-96)*, pages 79–86.

Jason Eisner. 1996c. Three New Probabilistic Models for Dependency Parsing: An Exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345.

Jason Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In Harry Bunt and Anton Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, pages 29–62. Kluwer Academic Publishers.

Jason Eisner and Noah Smith. 2005. Parsing with Soft and Hard Constraints on Dependency Length. In *Proceedings of the Ninth International Workshop on Parsing Technology (IWPT-05)*, pages 30–41.

Jakob Elming, Anders Johannsen, Sigrid Klerke, Emanuele Lapponi, Hector Martinez Alonso, and Anders Søgaard. 2013. Down-stream effects of tree-to-dependency conversions. In *Proceedings of the 2013 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-13)*, pages 617–626.

Timothy A. D. Fowler and Gerald Penn. 2010. Accurate Context-Free Parsing with Combinatory Categorial Grammar. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10)*, pages 335–344.

Daniel Gildea. 2001. Corpus Variation and Parser Performance. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing (EMNLP-01)*, pages 167–202.

Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING-12)*, pages 959–976.

Yoav Goldberg and Jon Orwant. 2013. A Dataset of Syntactic-Ngrams over Time from a Very Large Corpus of English Books. In *Proceedings of the 2nd Joint Conference on Lexical and Computational Semantics (*SEM-13)*, pages 241–247.

Jan Hajič. 1998. Building a Syntactically Annotated Corpus: The Prague Dependency Treebank. In Eva Hajičová, editor, *Issues of Valency and Meaning. Studies in Honor of Jarmila Panevová*, pages 12–19. Prague Karolinum, Charles University Press.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. 2009. The CoNLL-2009 Shared Task: Syntactic and Semantic Dependencies in Multiple Languages. In *Proceedings of the 13th Conference on Computational Natural Language Learning (CoNLL-09)*, pages 1–18.

Donald Hindle. 1983. User manual for Fidditch. Technical report, Naval Research Laboratory.

Julia Hockenmaier. 2003a. *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*. Ph.D. thesis, School of Informatics, University of Edinburgh, Edinburgh, United Kingdom.

Julia Hockenmaier. 2003b. Parsing with Generative Models of Predicate-Argument Structure. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL-03)*, pages 359–366.

Julia Hockenmaier. 2006. Creating a CCGbank and a Wide-Coverage CCG Lexicon for German. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING/ACL-06)*, pages 505–512.

Julia Hockenmaier and Yonatan Bisk. 2010. Normal-form parsing for Combinatory Categorial Grammars with generalized composition and type-raising. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*, pages 465–473.

Julia Hockenmaier and Mark Steedman. 2002. Generative Models for Statistical Parsing with Combinatory Categorial Grammar. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL-02)*, pages 335–342.

Julia Hockenmaier and Mark Steedman. 2005. CCGbank: Users' Manual. Technical Report MS-CIS-05-09, University of Pennsylvania Department of Computer and Information Science, Philadelphia, Pennsylvania, USA.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Matthew Honnibal. 2010. *Hat Categories: Representing Form and Function Simultaneously in Combinatory Categorial Grammar*. Ph.D. thesis, University of Sydney.

Eduard Hovy, Mitchell Marcus, Martha Palmer, Lance Ramshaw, and Ralph Weischedel. 2006. OntoNotes: The 90% Solution. In *Proceedings of the 2006 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-06)*, pages 57–60.

Liang Huang and David Chiang. 2005. Better k-best Parsing. In *Proceedings of the 9th International Workshop on Parsing Technology (IWPT-05)*, pages 53–64.

Liang Huang, Kevin Knight, and Aravind K. Joshi. 2006. Statistical Syntax-Directed Translation with Extended Domain of Locality. In *Proceedings of the 7th Biennial Conference of the Association for Machine Translation in the Americas (AMTA-06)*, pages 66–73.

Richard Johansson and Pierre Nugues. 2007. Extended Constituent-to-dependency Conversion for English. In *Proceedings of the 16th Nordic Conference of Computational Linguistics (NODALIDA-07)*, pages 105–112.

Mark Johnson and Ahmet Engin Ural. 2010. Reranking the Berkeley and Brown Parsers. In *Proceedings of the 2010 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-10)*, pages 665–668.

John Judge, Aoife Cahill, and Josef van Genabith. 2006. Questionbank: Creating a corpus of parse-annotated questions. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING/ACL-06)*, pages 497–504.

Tadao Kasami. 1965. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachusetts, USA.

Sunghwan Mac Kim, Dominick Ng, Mark Johnson, and James R. Curran. 2012. Improving Combinatory Categorial Grammar Parse Reranking with Dependency Grammar Features. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING-12)*, pages 1441–1458.

Tracy Holloway King, Richard Crouch, Stefan Riezler, Mary Dalrymple, and Ronald M. Kaplan. 2003. The PARC 700 Dependency Bank. In *Proceedings of the 4th International Workshop on Linguistically Interpreted Corpora (LINC-03)*, pages 1–8.

Dan Klein and Christopher D. Manning. 2003a. A* Parsing: Fast Exact Viterbi Parse Selection. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-03)*, pages 40–47.

Dan Klein and Christopher D. Manning. 2003b. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL-03)*, pages 423–430.

Dan Klein and Christopher D. Manning. 2003c. Fast Exact Inference with a Factored Model for Natural Language Parsing. In *Proceedings of the 17th Annual Conference on Neural Information Processing Systems (NIPS-03)*, pages 3–10.

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple Semi-supervised Dependency Parsing. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL-08)*, pages 595–603.

Terry Koo and Michael Collins. 2010. Efficient Third-Order Dependency Parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10)*, pages 1–11.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. 2009. *Dependency Parsing*. Morgan and Claypool.

H. Kucera and W. N. Francis. 1967. *Computational analysis of present-day American English*. Brown University Press.

Taku Kudo and Yuji Matsumoto. 2000. Japanese Dependency Structure Analysis Based on Support Vector Machines. In *Proceedings of the 2000 Conference on Empirical Methods in Natural Language Processing (EMNLP-00)*, pages 18–25.

Jonathan K. Kummerfeld, David Hall, James R. Curran, and Dan Klein. 2012. Parser Showdown at the Wall Street Corral: An Empirical Investigation of Error Types in Parser Output. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL-12)*, pages 1048–1059.

Jonathan K. Kummerfeld, Jessika Roesner, Tim Dawborn, James Haggerty, James R. Curran, and Stephen Clark. 2010. Faster Parsing by Supertagger Adaptation. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10)*, pages 345–355.

Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. 2013. Scaling Semantic Parsers with On-the-Fly Ontology Matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP-13)*, pages 1545–1556.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical Generalization in CCG Grammar Induction for Semantic Parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP-11)*, pages 1512–1523.

Mirella Lapata and Frank Keller. 2004. The Web as a Baseline: Evaluating the Performance of Unsupervised Web-based Models for a Range of NLP Tasks. In *Proceedings of the 2004 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-04)*, pages 121–128.

Tao Lei, Yu Xin, Yuan Zhang, Regina Barzilay, and Tommi Jaakkola. 2014. Low-Rank Tensors for Scoring Dependency Structures. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 1381–1391.

Mike Lewis and Mark Steedman. 2013. Combined Distributional and Logical Semantics. *Transactions of the Association for Computational Linguistics*, 1(2):179–192.

Mike Lewis and Mark Steedman. 2014a. A* CCG Parsing with a Supertag-factored Model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP-14)*, pages 990–1000.

Mike Lewis and Mark Steedman. 2014b. Improved CCG Parsing with Semi-supervised Supertagging. *Transactions of the Association for Computational Linguistics*, 2(1):327–338.

Dekang Lin. 1995. A Dependency-based Method for Evaluating Broad-Coverage Parsers. In *Proceedings of the 14th International Joint Conference on AI (IJCAI-95)*, pages 1420–1425.

Dekang Lin, Kenneth Church, Heng Ji, Satoshi Sekine, David Yarowsky, Shane Bergsma, Kailash Patil, Emily Pitler, Rachel Lathbury, Vikram Rao, Kapil Dalwani, and Sushant Narsale. 2010. New Tools for Web-Scale N-grams. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC-10)*.

Yuri Lin, Jean-Baptiste Michel, Erez Aiden Lieberman, Jon Orwant, Will Brockman, and Slav Petrov. 2012. Syntactic Annotations for the Google Books NGram Corpus. In *Proceedings of the ACL 2012 System Demonstrations*, pages 169–174.

Ji Ma, Yue Zhang, and Jingbo Zhu. 2014. Punctuation Processing for Projective Dependency Parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 791–796.

David Magerman. 1994. *Natural Language Parsing As Statistical Pattern Recognition*. Ph.D. thesis, Stanford University, Stanford, CA, USA.

David Magerman. 1995. Statistical Decision Tree Models for Parsing. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL-95)*, pages 276–283.

Christopher D. Manning and Bob Carpenter. 2000. Probabilistic Parsing Using Left Corner Language Models. In Harry Bunt and Anton Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, volume 16, pages 105–124. Kluwer Academic Publishers.

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

Svetoslav Marinov and Joakim Nivre. 2005. A Data-Driven Parser for Bulgarian. In *Proceedings of the Fourth Workshop on Treebanks and Linguistic Theories*, pages 89–100.

André Martins, Noah Smith, Eric Xing, Pedro Aguiar, and Mário Figueiredo. 2010. Turbo Parsers: Dependency Parsing by Approximate Variational Inference. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing (EMNLP-10)*, pages 34–44.

David McClosky, Eugene Charniak, and Mark Johnson. 2006. Effective Self-Training for Parsing. In *Proceedings of the 2006 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-06)*, pages 152–159.

David McClosky, Wanxiang Che, Marta Recasens, Mengqiu Wang, Richard Socher, and Christopher D. Manning. 2012. Stanford's System for Parsing the English Web. In *Notes of the First Workshop on the Syntactic Analysis of Non-Canonical Language (SANCL-12)*.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005a. Online Large-Margin Training of Dependency Parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL-05)*, pages 91–98.

Ryan McDonald and Joakim Nivre. 2011. Analyzing and Integrating Dependency Parsers. *Computational Linguistics*, 37(1):197–230.

Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. 2013. Universal Dependency Annotation for Multilingual Parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*, pages 92–97.

Ryan McDonald and Fernando Pereira. 2006. Online Learning of Approximate Dependency Parsing Algorithms. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL-06)*, pages 81–88.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005b. Non-Projective Dependency Parsing using Spanning Tree Algorithms. In *Proceedings of the 2005 Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP-05)*, pages 523–530.

Stephen Merity and James R. Curran. 2011. Frontier Pruning for Shift-Reduce CCG Parsing. In *Proceedings of the Australasian Language Technology Association Workshop 2011 (ALTW-11)*, pages 66–75.

Adam Meyers, Ruth Reeves, Catherine Macleod, Rachel Szekely, Veronika Zielinska, Brian Young, and Ralph Grishman. 2004. The NomBank Project: An Interim Report. In *Proceedings of the NAACL/HLT Workshop on Frontiers in Corpus Annotation*, pages 24–31.

Jean-Baptiste Michel, Yuan K. Shen, Aviva P. Aiden, Adrian Veres, Matthew K. Gray, The Google Books Team, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin A. Nowak, and Erez L. Aiden. 2011. Quantitative Analysis of Culture Using Millions of Digitized Books. *Science*, 331(6014):176–182.

Yusuke Miyao, Takashi Ninomiya, and Jun'ichi Tsujii. 2004. Corpus-oriented Grammar Development for Acquiring a Head-driven Phrase Structure Grammar from the Penn Treebank. In *Proceedings of the 1st International Joint Conference on Natural Language Processing (IJCNLP-04)*, pages 684–693.

Yusuke Miyao and Jun'ichi Tsujii. 2002. Maximum Entropy Estimation for Feature Forests. In *Proceedings of the Second International Conference on Human Language Technology Research (HLT-02)*, pages 292–297.

Preslav Nakov and Marti Hearst. 2005a. Search Engine Statistics Beyond the n-Gram: Application to Noun Compound Bracketing. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-05)*, pages 17–24.

Preslav Nakov and Marti Hearst. 2005b. Using the Web as an Implicit Training Set: Application to Structural Ambiguity Resolution. In *Proceedings of the 2005 Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP-05)*, pages 835–842.

Dominick Ng. 2010. Improved Evaluation and Parse Reranking for Combinatory Categorial Grammar. Honours thesis, University of Sydney, Sydney, Australia.

Dominick Ng, Mohit Bansal, and James R. Curran. 2015. Web-scale Surface and Syntactic n-gram Features for Dependency Parsing. arXiv:1502.07038.

Dominick Ng and James R. Curran. 2012. Dependency Hashing for n-best CCG Parsing. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL-12)*, pages 497–505.

Dominick Ng and James R. Curran. 2015. Identifying Cascading Errors using Constraints in Dependency Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 1148–1158.

Dominick Ng, Matthew Honnibal, and James R. Curran. 2010. Reranking a wide-coverage CCG parser. In *Proceedings of the Australasian Language Technology Association Workshop 2010 (ALTW-10)*, pages 90–98.

Jens Nilsson. 2009. *Transformation and Combination in Data-Driven Dependency Parsing*. Ph.D. thesis, Växjö University.

Joakim Nivre. 2006. *Inductive Dependency Parsing*. Springer.

Joakim Nivre. 2009. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP (ACL/IJCNLP-09)*, pages 351–359.

Joakim Nivre, Yoav Goldberg, and Ryan McDonald. 2014. Constrained arc-eager dependency parsing. *Computational Linguistics*, 40(2):249–257.

Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 Shared Task on Dependency Parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-Based Dependency Parsing. In *Proceedings of the Eighth Conference on Computational Natural Language Learning (CoNLL-04)*, pages 49–56.

Joakim Nivre, Laura Rimell, Ryan McDonald, and Carlos Gómez Rodríguez. 2010. Evaluation of Dependency Parsers on Unbounded Dependencies. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*, pages 833–841.

Joakim Nivre and Mario Scholz. 2004. Deterministic Dependency Parsing of English Text. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING-04)*, pages 64–70.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The Proposition Bank: An Annotated Corpus of Semantic Roles. *Computational Linguistics*, 31(1):71–105.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning Accurate, Compact, and Interpretable Tree Annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING/ACL-06)*, pages 433–440.

Slav Petrov and Dan Klein. 2007. Improved Inference for Unlexicalized Parsing. In *Proceedings of the 2007 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-07)*, pages 404–411.

Slav Petrov and Ryan McDonald. 2012. Overview of the 2012 Shared Task on Parsing the Web. In *Notes of the First Workshop on the Syntactic Analysis of Non-Canonical Language (SANCL-12)*.

Emily Pitler. 2012a. Attacking Parsing Bottlenecks with Unlabeled Data and Relevant Factorizations. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL-12)*, pages 768–776.

Emily Pitler. 2012b. Conjunction Representation and Ease of Domain Adaptation. In *Notes of the First Workshop on the Syntactic Analysis of Non-Canonical Language (SANCL-12)*.

Emily Pitler, Shane Bergsma, Dekang Lin, and Kenneth Church. 2010. Using Web-scale N-grams to Improve Base NP Parsing Performance. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*, pages 886–894.

Sampo Pyysalo, Filip Ginter, Veronika Laippala, Katri Haverinen, Juho Heimonen, and Tapio Salakoski. 2007. On the unification of syntactic annotations under the Stanford dependency scheme: A case study on BioInfer and GENIA. In *Proceedings of the Biological, Translational, and Clinical Language Processing Workshop*, pages 25–32.

Adwait Ratnaparkhi. 1996. A Maximum Entropy Model for Part-of-Speech Tagging. In *Proceedings of the 1996 Conference on Empirical Methods in Natural Language Processing (EMNLP-96)*, pages 133–142.

Adwait Ratnaparkhi. 1997. A Linear Observed Time Statistical Parser Based on Maximum Entropy Models. In *Proceedings of the 1997 Conference on Empirical Methods in Natural Language Processing (EMNLP-97)*, pages 1–10.

Siva Reddy, Mirella Lapata, and Mark Steedman. 2014. Large-scale Semantic Parsing without Question-Answer Pairs. *Transactions of the Association for Computational Linguistics*, 2(1):377–392.

Laura Rimell and Stephen Clark. 2008. Adapting a Lexicalized-Grammar Parser to Contrasting Domains. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP-08)*, pages 475–484.

Laura Rimell, Stephen Clark, and Mark Steedman. 2009. Unbounded Dependency Recovery for Parser Evaluation. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP-09)*, pages 813–821.

Brian Roark and Kristy Hollingshead. 2008. Classifying Chart Cells for Quadratic Complexity Context-Free Inference. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*, pages 745–752.

Brian Roark and Kristy Hollingshead. 2009. Linear Complexity Context-Free Parsing Pipelines via Chart Constraints. In *Proceedings of the 2009 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-09)*, pages 647–655.

Alexander Rush and Slav Petrov. 2012. Vine Pruning for Efficient Multi-Pass Dependency Parsing. In *Proceedings of the 2012 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL-12)*, pages 498–507.

Kenji Sagae and Jun'ichi Tsujii. 2007. Dependency Parsing and Domain Adaptation with LR Models and Parser Ensembles. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 1044–1050.

Anoop Sarkar. 2001. Applying Co-Training Methods to Statistical Parsing. In *Proceedings of the 2nd Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 95–102.

Satoshi Sekine. 1997. The Domain Dependence of Parsing. In *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP-97)*, pages 96–102.

Libin Shen, Lucas Champollion, and Aravind K. Joshi. 2008. LTAG-spinal and the Treebank. *Language Resources and Evaluation*, 42(1):1–19.

Daniel D. Sleator and Davy Temperley. 1991. Parsing English with a Link Grammar. Technical Report CMU-CS-91-196, Carnegie Mellon University.

Anders Søgaard. 2013. An Empirical Study of Differences between Conversion Schemes and Annotation Guidelines. In *Proceedings of the 2nd International Conference on Dependency Linguistics (DEPLING-13)*, pages 298–307.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press, Cambridge, Massachusetts, USA.

Mark Steedman, Anoop Sarkar, Miles Osborne, Rebecca Hwa, Stephen Clark, Julia Hockenmaier, Paul Ruhlen, Steven Baker, and Jeremiah Crim. 2003. Bootstrapping Statistical Parsers from Small Datasets. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics (EACL-03)*, pages 331–338.

Will Styler. 2011. The EnronSent Corpus. Technical Report 01-2011, University of Colorado at Boulder, Boulder, Colorado.

Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. 2008. The CoNLL 2008 Shared Task on Joint Parsing of Syntactic and Semantic Dependencies. In *Proceedings of the 12th Conference on Computational Natural Language Learning (CoNLL-08)*, pages 159–177.

Daniel Tse and James R. Curran. 2010. Chinese CCGbank: extracting CCG derivations from the Penn Chinese Treebank. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*, pages 1083–1091.

David Vadas and James R. Curran. 2007. Adding Noun Phrase Structure to the Penn Treebank. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL-07)*, pages 240–247.

David Vadas and James R. Curran. 2008. Parsing Noun Phrase Structure with CCG. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL-08)*, pages 335–343.

K. Vijay-Shanker and David J. Weir. 1994. The Equivalence of Four Extensions of Context-Free Grammars. *Mathematical Systems Theory*, 27(6):511–546.

Aline Villavicencio. 2002. Learning to Distinguish PP Arguments from Adjuncts. In *Proceedings of the 6th Conference on Natural Language Learning (CoNLL-02)*, pages 84–90.

Martin Volk. 2001. Exploiting the WWW as a corpus to resolve PP attachment ambiguities. In *Proceedings of the Corpus Linguistics 2001 Conference (CL-01)*, pages 601–606.

Colin Warner, Ann Bies, Christine Brisson, and Justin Mott. 2004. Addendum to the Penn Treebank II Style Bracketing Guidelines: BioMedical Treebank Annotation. Technical report, University of Pennsylvania.

Ralph Weischedel, Eduard Hovy, Mitchell Marcus, Martha Palmer, Robert Belvin, Sameer Pradhan, Lance Ramshaw, and Nianwen Xue. 2011. OntoNotes: A Large Training Corpus for Enhanced Processing. In Joseph Olive, Caitlin Christianson, and John McCary, editors, *Handbook of Natural Language Processing and Machine Translation: DARPA Global Autonomous Language Exploitation*, pages 54–63. Springer.

Michael White and Rajakrishnan Rajkumar. 2008. A More Precise Analysis of Punctuation for Broad-Coverage Surface Realization with CCG. In *Proceedings of the COLING 2008 Workshop on Grammar Engineering Across Frameworks*, pages 17–24.

Kent Wittenburg. 1986. *Natural Language Parsing with Combinatory Categorial Grammars in a Graph-Unification-Based Formalism*. Ph.D. thesis, University of Texas at Austin.

Kent Wittenburg. 1987. Predictive Combinators: A Method for Efficient Processing of Combinatory Categorial Grammars. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics (ACL-87)*, pages 73–80.

Wenduan Xu, Stephen Clark, and Yue Zhang. 2014. Shift-Reduce CCG Parsing with a Dependency Model. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 218–227.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical Dependency Analysis with Support Vector Machines. In *Proceedings of the 8th International Workshop of Parsing Technologies (IWPT-03)*, pages 196–206.

Daniel H. Younger. 1967. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2):189–208.

Hao Zhang and Ryan McDonald. 2014. Enforcing Structural Diversity in Cube-pruned Dependency Parsing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 656–661.

Meishan Zhang, Wanxiang Che, Yijia Liu, Zhenghua Li, , and Ting Liu. 2012. HIT Dependency Parsing: Bootstrap Aggregating Heterogeneous Parsers. In *Notes of the First Workshop on the Syntactic Analysis of Non-Canonical Language (SANCL-12)*.

Yue Zhang, Byung-Gyu Ahn, Stephen Clark, Curt Van Wyk, James R. Curran, and Laura Rimell. 2010. Chart Pruning for Fast Lexicalised-Grammar Parsing. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING-10)*, pages 1471–1479.

Yue Zhang and Stephen Clark. 2011a. Shift-Reduce CCG Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-11)*, pages 683–692.

Yue Zhang and Stephen Clark. 2011b. Syntactic Processing Using the Generalized Perceptron and Beam Search. *Computational Linguistics*, 37(1):105–151.

Yue Zhang and Joakim Nivre. 2011. Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-11)*, pages 188–193.