THE UNIVERSITY
of EDINBURGH

Transition-based Combinatory Categorial Grammar parsing for English and Hindi

Bharat Ram Ambati

Doctor of Philosophy

Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh

2016

Abstract

Given a natural language sentence, parsing is the task of assigning it a grammatical structure, according to the rules within a particular grammar formalism. Different grammar formalisms like Dependency Grammar, Phrase Structure Grammar, Combinatory Categorial Grammar, Tree Adjoining Grammar are explored in the literature for parsing. For example, given a sentence like "John ate an apple", parsers based on the widely used dependency grammars find grammatical relations, such as that 'John' is the subject and 'apple' is the object of the action 'ate'. We mainly focus on Combinatory Categorial Grammar (CCG) in this thesis.

In this thesis, we present an incremental algorithm for parsing CCG for two diverse languages: English and Hindi. English is a fixed word order, SVO (Subject-Verb-Object), and morphologically simple language, whereas, Hindi, though predominantly a SOV (Subject-Object-Verb) language, is a free word order and morphologically rich language. Developing an incremental parser for Hindi is really challenging since the predicate needed to resolve dependencies comes at the end. As previously available shift-reduce CCG parsers use English CCGbank derivations which are mostly right branching and non-incremental, we design our algorithm based on the dependencies resolved rather than the derivation. Our novel algorithm builds a dependency graph in parallel to the CCG derivation which is used for revealing the unbuilt structure without backtracking. Though we use dependencies for meaning representation and CCG for parsing, our revealing technique can be applied to other meaning representations like lambda expressions and for non-CCG parsing like phrase structure parsing.

Any statistical parser requires three major modules: data, parsing algorithm and learning algorithm. This thesis is broadly divided into three parts each dealing with one major module of the statistical parser. In Part I, we design a novel algorithm for converting dependency treebank to CCGbank. We create Hindi CCGbank with a decent coverage of 96% using this algorithm. We also do a cross-formalism experiment where we show that CCG supertags can improve widely used dependency parsers. We experiment with two popular dependency parsers (Malt and MST) for two diverse languages: English and Hindi. For both languages, CCG categories improve the overall accuracy of both parsers by around 0.3-0.5% in all experiments. For both parsers, we see larger improvements specifically on dependencies at which they are known to be weak: long distance dependencies for Malt, and verbal arguments for MST. The result is particularly interesting in the case of the fast greedy parser (Malt), since

improving its accuracy without significantly compromising speed is relevant for large scale applications such as parsing the web.

We present a novel algorithm for incremental transition-based CCG parsing for English and Hindi, in Part II. Incremental parsers have potential advantages for applications like language modeling for machine translation and speech recognition. We introduce two new actions in the shift-reduce paradigm for revealing the required information during parsing. We also analyze the impact of a beam and look-ahead for parsing. In general, using a beam and/or look-ahead gives better results than not using them. We also show that the incremental CCG parser is more useful than a non-incremental version for predicting relative sentence complexity. Given a pair of sentences from wikipedia and simple wikipedia, we build a classifier which predicts if one sentence is simpler/complex than the other. We show that features from a CCG parser in general and incremental CCG parser in particular are more useful than a chart-based phrase structure parser both in terms of speed and accuracy.

In Part III, we develop the first neural network based training algorithm for parsing CCG. We also study the impact of neural network based tagging models, and greedy versus beam-search parsing, by using a structured neural network model. In greedy settings, neural network models give significantly better results than the perceptron models and are also over three times faster. Using a narrow beam, structured neural network model gives consistently better results than the basic neural network model. For English, structured neural network gives similar performance to structured perceptron parser. But for Hindi, structured perceptron is still the winner.

Lay Summary

Given a natural language sentence, the task of finding its grammatical structure in a grammar formalism is called natural language parsing. For example, given a sentence like "John ate an apple", parsers based on the widely used dependency grammars find grammatical relations, such as that 'John' is the subject and 'apple' is the object of the action 'ate'. We mainly focus on a grammar formalism called Combinatory Categorial Grammar (CCG) in this thesis.

In this thesis, we present an incremental algorithm for parsing CCG for two diverse languages: English and Hindi. English is a fixed word order and morphologically simple language, whereas, Hindi is a free word order and morphologically rich language. Our algorithm builds a dependency graph in parallel to the CCG derivation which is used for revealing the unbuilt structure without backtracking. Though we use dependencies for meaning representation and CCG for parsing, our revealing technique can be applied to other meaning representations like lambda expressions and for non-CCG parsing like phrase structure parsing.

This thesis is broadly divided into three parts each dealing with a major module of the statistical parser: data, parsing algorithm and learning algorithm. In Part I, we design a novel algorithm for converting the Hindi dependency treebank to a Hindi CCGbank. We also do a cross-formalism experiment where we show that CCG supertags can improve widely used dependency parsers. We experiment with two popular dependency parsers (Malt and MST) for two diverse languages (English and Hindi) and show that CCG categories improve the accuracy by around 0.3-0.5% in all experiments.

We present a novel algorithm for incremental transition-based CCG parsing for English and Hindi, in Part II. We introduce two new actions in the shift-reduce paradigm for revealing the required information during parsing. We also analyze the impact of a beam and look-ahead for parsing. In general, using a beam and/or look-ahead gives better results than not using them. We also show that incremental CCG parser is more useful than a non-incremental parser for predicting relative sentence complexity.

In Part III, we develop the first neural network based training algorithm for parsing CCG. We also study the impact of neural network based tagging models, and greedy versus beam-search parsing, by using a structured neural network model. In greedy settings, neural network models give significantly better results than the perceptron models and are also over three times faster. Using a (narrow) beam, structured neural network model gives consistently better results than the basic neural network model.

Acknowledgements

I would like to thank my primary supervisor Prof. Mark Steedman for his invaluable guidance. This thesis would not have been possible without his support. In addition to suggesting new ideas he gave full freedom to explore new paths. I learnt a lot from his immense knowledge in both linguistics and computational approaches. Tejaswini, my secondary supervisor was really helpful during the early stages of my PhD. Trained in linguistics herself, I learnt about Hindi linguistics from her. Thanks to Mark Johnson for the discussions about incremental parsing and feedback on the paper.

ILCC is one of the best places to do a PhD in NLP and I loved every single day of my work here. There are a lot of academic and fun activities through out the year. Friday seminars, ProbModels and ML for NLP reading groups, Language lunches, speaker lunches are a few of them. Thanks to Steedman gang: Greg ("the expert"), Mike ("the lewiser"), Aciel ("Dr. Eshky"), mini Mark, Christos, Siva, John, Andrew, Omri, Kira, Nathanial, Lexi, Jeff and Nathan for making the weekly meetings interesting, feedback on my papers and discussions about CCG, perceptron etc. I had wonderful time with other ILCC friends and faculty: Carina, Michael, Lea, Dominikus, Herman, Janie, Philip, Maria, Annie, Shashi, Ben, Diego, Des, Eva, Stella, Dave, Francesco, Sharon, Mirella, Adam, Shay, and Frank. Thanks to level 3 admin staff: Julie, Diana, Avril and Nicola for helping out with all the admin related things. Special thanks to Tom Kwiatkowski and Julia Hockenmaier for suggestions on incremental parsing and Joakim Nivre for the invited talk at Uppsala. Thanks to Shay Cohen and Steve Clark for being my examiners and giving wonderful feedback on this thesis.

I had such a great time in Edinburgh. Being miles away from home, my friends here made me feel home. I would like to acknowledge my friends Siva Reddy, Spandana, Gangireddy, Srikanth, Praveen, Rupa who are there with me throughout my PhD journey. It is nice to have old friends like Siva and Spandana whom I know for nearly 10 years. Thanks to Manic, Ramya, Bhargav, Siva Prakash, Rajkarn, Sameer for nice discussions. Special thanks to 3.39 office gang, Aciel, Greg, Mike, mini Mark, Christos, Victor, Janie, Akash, Stef, and Philip. In addition to Informatics gang, I enjoyed my time with the Edinburgh Bhangra Crew, specially Tanvi, Kajol, Aneesa, Mani, Laura.

I had a chance to work on real-world problems during my three months internship at Apple Siri speech team. Thanks to Mahesh, Sameer, Venki and Ernie at Apple Siri team for introducing me to the corporate world.

Finally, I would like to thank my father, mother, grand mother and sister for their support and strong belief in me.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Bharat Ram Ambati*)

Table of Contents

List of Figures

List of Tables

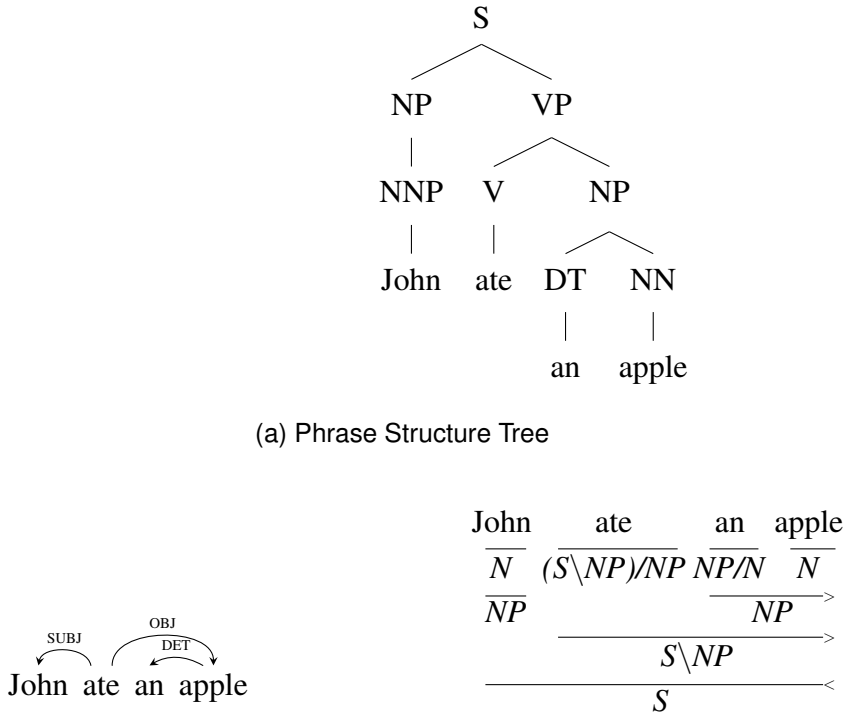
Chapter 1

Introduction

Given a natural language sentence, parsing is the task of assigning it a grammatical structure, according to the rules within a particular grammar formalism. Different grammar formalisms like Dependency Grammar, Phrase Structure Grammar, Combinatory Categorial Grammar (CCG), Tree Adjoining Grammar are explored in the literature for parsing. For example, given a sentence like "John ate an apple", parsers based on the widely used dependency grammars find grammatical relations, such as that 'John' is the subject and 'apple' is the object of the action 'ate'. Figure 1.1 presents the parse trees for this example sentence in phrase structure, dependency and combinatory categorial grammar formalisms.

Parsing is one of the major tasks which helps in understanding the natural language. It is useful for several real-world applications like machine translation (Galley et al., 2006; Quirk and Menezes, 2006; Katz-Brown et al., 2011; Sennrich, 2015), question answering (Kwiatkowski et al., 2013; Reddy et al., 2014, 2016), dialogue systems (Stoness et al., 2004), and speech recognition (Chelba and Jelinek, 2000; Roark, 2001). Parsers can be grammar-driven, data-driven/statistical or hybrid. Statistical parsers differ from grammar driven parsers as they use a corpus to induce a probabilistic model for disambiguation. Statistical parsers can be broadly divided into graph-based parsers (McDonald et al., 2005b; Clark and Curran, 2007) and transition-based parsers (Nivre et al., 2007b; Zhang and Clark, 2011a). Graph-based parsers use exhaustive global search resulting in a stronger and more accurate model. However, transition-based parsers are more appealing for practical real-time applications since parsing can be achieved in linear time compared to graph-based parsers, whose complexity is generally polynomial in time.

In this thesis we focus on transition-based incremental parsing for Combinatory



(a) Phrase Structure Tree

(b) Dependency Tree

(c) CCG Derivation Tree

Figure 1.1: Parse trees in different grammar formalisms.

Categorial Grammar formalism. Incremental parsers are cognitively more plausible than non-incremental parsers and are more appealing for applications like real-time statistical machine translation and speech recognition. We work with two typologically diverse languages: English and Hindi. In this chapter, we first list the major contributions of this thesis. We give a brief introduction to Combinatory Categorial Grammar. Then we describe our approach for developing transition-based incremental CCG parsers. Finally, we present the outline of this thesis.

1.1 Contributions

The major contributions of this thesis are

- Building on earlier work by Cakici (2005), we present an algorithm for converting dependency treebanks into CCGbanks.
- We show that informative CCG categories improve the performance of widely used dependency parsers like Malt and MST.

- We develop a new incremental algorithm for parsing CCG by introducing two new actions which reveal unbuilt structure from overly greedy prefix analyses.
- For the task of predicting relative sentence complexity, we show that an incremental CCG parser gives significant improvements in speed and accuracy compared to a phrase structure parser.
- We present different neural network models for transition-based CCG parsing, the first neural network based parsers for CCG [1].
- For all of the above (except for 1 and 4), we present experiments and results in two typologically diverse languages: English (SVO word order) and Hindi (SOV word order)

1.2 Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG) (Steedman, 2000) is a strongly lexicalized grammar formalism, in the sense that all language-specific information including linear order is defined at the level of the lexicon. It is "nearly context-free" in expressive power, in the sense of being among a group of formalisms for natural language grammars that are at the least more expressive linguistically significant level than context-free grammar (CFG) (Joshi et al., 1991). It has a completely type-transparent interface between syntactic derivation and compositional assembly of the underlying semantic representation, including predicate argument structure, quantification and information structure. Because of this semantic transparency, CCG is widely used in practical applications involving semantic interpretation and inference (Bos et al., 2004; Lewis and Steedman, 2013a,b) especially for semantic parsing with special focus on question answering (Kwiatkowski et al., 2013; Reddy et al., 2014).

In the categorial lexicon, words are associated with syntactic categories, such as $S\backslash NP$ or $(S\backslash NP)/NP$ for English intransitive and transitive verbs. Categories of the form $X\backslash Y$ or X/Y are functors, which take an argument Y to their left or right (depending on the direction of the slash) and yield a result X. Every syntactic category is paired with a semantic interpretation (usually expressed as a λ-term).

Like all variants of categorial grammar, CCG uses function application to combine constituents, but it also uses a set of linear order-dependent syntactic combinatory

[1] At the same time, and independent of this thesis, Xu et al. (2016) developed a neural network based CCG parser.

rules corresponding semantically to composition (**B**) and type-raising (**T**). Type raising is a non-recursive lexical operation related to (abstract) case. However, for fixed word order languages without morphological case, Hockenmaier and Steedman (2007) advocate the use of unary type-changing rules for reasons of efficiency, including type-raising rules and additional rules to deal with complex adjunct categories (e.g $(NP\backslash NP) \Longrightarrow S[ng]\backslash NP$ for ing-VPs that act as noun phrase modifiers). Examples of CCG rules are:

Forward Application ($>$):	X/Y	Y	$\Longrightarrow$	X
Backward Application ($<$):	Y	X\Y	$\Longrightarrow$	X
Forward Composition ($>$ B):	X/Y	Y/Z	$\Longrightarrow$	X/Z
Backward Composition ($<$ B):	Y\Z	X\Y	$\Longrightarrow$	X\Z
Forward Crossed Composition ($> B_X$):	X/Y	Y\Z	$\Longrightarrow$	X\Z
Backward Crossed Composition ($< B_X$):	Y/Z	X\Y	$\Longrightarrow$	X/Z
Forward Type-raising ($>$ T) :	X		$\Longrightarrow$	T/(T\X)
Backward Type-raising ($<$ T) :	X		$\Longrightarrow$	T\(T/X)

1.3 Problem and Approach

We present an incremental algorithm for parsing Combinatory Categorial Grammar (CCG) for two diverse languages: English and Hindi. English is a fixed word order and morphologically simple language, whereas Hindi is a free word order and morphologically rich language. The treebank available for Hindi is four times smaller than the English treebank making the task of statistical Hindi parsing more challenging. Any statistical parser requires three major modules: data, parsing algorithm and learning algorithm. This thesis is broadly divided into three parts each dealing with one major module of the statistical parser.

1.3.1 Data

Statistical parsers require data to train the model. To develop statistical CCG parsers, we need resources like CCGbanks (Hockenmaier and Steedman, 2007) or CCG lexicons (Çakıcı, 2009). Hockenmaier and Steedman (2007) developed the first English CCGbank automatically from the Penn Wall Street Journal Phrase Structure Treebank (Marcus et al., 1993). Availability of the English CCGbank has enabled the creation of several robust and accurate wide-coverage CCG parsers for English, both graph-based (Hockenmaier and Steedman, 2002; Clark and Curran, 2007; Auli and

Lopez, 2011; Lewis and Steedman, 2014a) and transition-based (Zhang and Clark, 2011a; Xu et al., 2014), that are being used extensively for broad-coverage parsing, and especially for tasks requiring deep linguistic analysis such as semantic parsing and question-answering (Bos et al., 2004; Lewis and Steedman, 2013a,b; Kwiatkowski et al., 2013; Reddy et al., 2014). Unlike English, Hindi is a morphologically rich and free word order language. Creation of CCGbanks in other languages, especially languages typologically far from English is beneficial both for the development of CCG analyses for linguistic phenomena in these languages, and also for the development of deep NLP tools for these languages. There is no CCGbank available for Hindi and manual creation of such resources takes a lot of effort. But, a Hindi dependency treebank (Bhatt et al., 2009) was recently developed for Hindi. We present an approach for automatically creating a CCG treebank from a dependency treebank for Hindi. Rather than a direct conversion from dependency trees to CCG trees, we propose a two stage approach: a language independent generic algorithm first extracts a CCG lexicon from the dependency treebank. A deterministic CCG parser then creates a treebank of CCG derivations. The advantage of this approach is that we can handle crossing-arcs in the dependency treebanks in a better way.

After developing CCG resources like lexicon and supertagger for Hindi, we do a cross-formalism experiment where we show that CCG supertags can improve dependency parsing. Different grammar formalisms have different advantages and providing features from one formalism can be useful for parsing in another formalism (Sagae et al., 2007; Coppola and Steedman, 2013; Kim et al., 2012). CCG categories contain subcategorization information which is a useful feature in dependency parsing (Zhang and Nivre, 2011). We provide CCG categories as features to dependency parsers and show that CCG categories helps dependency parsing. We experiment with two popular dependency parsers: Malt[2] (Nivre et al., 2007b) and MST[3] (McDonald, 2006) and for two languages: English and Hindi. For both languages and both the parsers, CCG categories improve the overall accuracy by around 0.3-0.5% in all experiments.

1.3.2 Parsing Algorithm

Traditionally transition-based parsers use arc-eager or arc-standard style parsing algorithms (Nivre, 2003, 2004). Zhang and Clark (2011a) developed the first shift-reduce CCG parser for English which gave accuracies competitive with graph-based

[2] http://www.maltparser.org/

[3] http://mstparser.sourceforge.net/

parsers (Clark and Curran, 2007). Improving Zhang and Clark (2011a)'s model, Xu et al. (2014) developed a dependency model for shift-reduce CCG parsing using a dynamic oracle technique (Goldberg and Nivre, 2012). Shift-reduce CCG parsers rely either on normal-form (Eisner, 1996a) CCGbank derivations (Zhang and Clark, 2011a) which are non-incremental, or on dependencies (Xu et al., 2014) which could be incremental in simple cases, but do not guarantee incrementality. As these parsers employ an arc-standard (Yamada and Matsumoto, 2003) shift-reduce strategy on CCGbank, given an SVO language, these parsers are not guaranteed to attach the subject before the object.

Besides being cognitively plausible (Marslen-Wilson, 1973), incremental parsing is more useful than non-incremental parsing for some applications. For example, an incremental analysis is required for integrating syntactic and semantic information into language modeling for statistical machine translation (SMT) and automatic speech recognition (ASR) (Roark, 2001; Wang and Harper, 2003). We develop a new incremental shift-reduce algorithm for parsing CCG by building a dependency graph in addition to the CCG derivation as a representation. The dependencies in the graph are extracted from the CCG derivation. We introduce two new actions in the shift-reduce paradigm for "revealing" (Pareschi and Steedman, 1987) unbuilt structure during parsing. We build the dependency graph in parallel to the incremental CCG derivation and use this graph for revealing, via these two new actions. As our algorithm does not model derivations, but rather models transitions, we do not need a treebank of incremental CCG derivations and can train on the dependencies in the existing treebank. Our approach can therefore be adapted to other languages with dependency treebanks, since CCG lexical categories can be easily extracted from dependency treebanks (Cakici, 2005; Ambati et al., 2013). In this thesis, we use dependencies for meaning representation and CCG for parsing. But our revealing technique is generic enough and can be applied to other meaning representations like lambda expressions and for non-CCG parsing like phrase structure parsing. Similar to CCG derivation, we can extract dependencies from the phrase structure tree using head dependency rules (Collins, 1999; Johansson and Nugues, 2007; de Marneffe et al., 2006). Also, lambda expression can be represented in the form of a dependency graph similar to our representation (Kwiatkowski et al., 2010).

We experiment with both English and Hindi CCGbanks. Developing an incremental algorithm for a predominantly SOV language like Hindi is much more challenging than for an SVO language. Other factors like free word order nature, morphological

richness and less training data played a key role in developing incremental parsing models. We experimented with different levels of incrementality through experiments analyzing the impact of greedy vs. beam and with/without look-ahead.

We show the practical implications of incremental parsers using an application of predicting relative sentence complexity. Given a pair of sentences from Wikipedia and Simple Wikipedia, we build a classifier which predicts if one sentence is simpler/complex than the other. We experiment with features from different versions of incremental CCG parsers, non-incremental CCG parser and phrase structure parser. We show that CCG parse features in general and incremental CCG parser in particular is more useful than a chart-based phrase structure parser both in terms of speed and accuracy.

1.3.3 Learning Algorithm

Neural network models are gaining popularity because of both speed and accuracy. Recent neural network based parsing work resulted in state-of-the-art dependency parsers (Chen and Manning, 2014; Weiss et al., 2015; Alberti et al., 2015). Following these developments, we built neural network and structured neural network models for parsing CCG. In greedy settings, neural network models give significantly better results than the perceptron models and are also over three times faster. In the case of beam, structured neural network model gives consistently better results than the basic neural network model. These results are consistent across English and Hindi.

Lewis and Steedman (2014b) and Xu et al. (2015a) showed that neural network based supertaggers perform better than the maximum entropy based supertagger (Clark and Curran, 2004a) for the state-of-the-art graph based parsers like C&C (Clark and Curran, 2007) and EasyCCG (Lewis and Steedman, 2014a) parsers. Following this work, we show that neural network taggers give better results compared to maximum entropy based taggers for transition-based CCG parsing as well.

1.4 Thesis Outline

Chapter 2: Hindi CCGbank from Dependency Treebank In this chapter, we first present an approach for automatically creating a Combinatory Categorial Grammar (CCG) treebank from a dependency treebank for the Subject-Object-Verb language Hindi. Rather than a direct conversion from dependency trees to CCG

trees, we propose a two stage approach: a language independent generic algorithm first extracts a CCG lexicon from the dependency treebank. A deterministic CCG parser then creates a treebank of CCG derivations. We also discuss special cases of this generic algorithm to handle linguistic phenomena specific to Hindi. In doing so we extract different constructions with long-range dependencies like coordinate constructions and non-projective dependencies resulting from constructions like relative clauses, noun elaboration and verbal modifiers. The content of most of this chapter is published in Ambati et al. (2013) and Ambati et al. (2016a).

Chapter 3: Improving Dependency Parsers using CCG Supertags

Subcategorization information is a useful feature in dependency parsing. In this chapter, we explore a method of incorporating this information via CCG categories from a supertagger. We experiment with two popular dependency parsers (Malt and MST) for two languages: English and Hindi. For both languages, CCG categories improve the overall accuracy of both parsers by around 0.3-0.5% in all experiments. For both parsers, we see larger improvements specifically on dependencies at which they are known to be weak: long distance dependencies for Malt, and verbal arguments for MST. The result is particularly interesting in the case of the fast greedy parser (Malt), since improving its accuracy without significantly compromising speed is relevant for large scale applications such as parsing the web. Parts of this chapter are based on the content from Ambati et al. (2013) and Ambati et al. (2014).

Chapter 4: Incremental Parsing for English Incremental parsers have potential advantages for applications like language modeling for machine translation and speech recognition. We describe a new algorithm for incremental transition-based Combinatory Categorial Grammar parsing. As English CCGbank derivations are mostly right branching and non-incremental, we design our algorithm based on the dependencies resolved rather than the derivation. We introduce two new actions in the shift-reduce paradigm based on the idea of 'revealing' (Pareschi and Steedman, 1987) the required information during parsing. We present two versions of the incremental parser: a greedy parser which uses a look-ahead and a beam search parser which doesn't use a look-ahead. On the standard CCGbank test data, our greedy parser achieves improvements of 0.88% in labelled and 2.0% in unlabelled F-score over a greedy non-incremental shift-

reduce parser.

Chapter 5: Incremental Parsing for Hindi In this chapter, we present transition-based CCG parsers for Hindi. We first extend the Zhang and Clark (2011a)'s shift-reduce model by adding Hindi-specific features to build the first shift-reduce CCG parser for Hindi. We analyze the impact of different settings like chunk and morphological features, greedy vs. beam-search parsing, gold vs. automatic features, coarse-grained vs. fine-grained lexicon. With automatic features, beam-search parser with coarse-grained lexicon gave the best unlabelled and labelled F-scores of 85.60% and 77.32% respectively. Then we design an incremental algorithm extending the revealing based incremental algorithm presented in the previous chapter. We make several extensions to make the algorithm as incremental as possible.

Chapter 6: Assessing Relative Sentence Complexity using Incremental Parsers In this chapter, we see how incremental CCG parsers can help in a practical application like predicting the relative sentence complexity. Given a pair of sentences, we present computational models to assess if one sentence is simpler to read than the other. While existing models explored the usage of phrase structure features using a non-incremental parser, experimental evidence suggests that the human language processor works incrementally. We empirically evaluate if syntactic features from incremental CCG parsers are more useful than features from a non-incremental phrase structure parser. Our evaluation on Simple and Standard Wikipedia sentence pairs shows that incremental CCG parser gives significant improvements in speed (12 times faster) as well as in terms of accuracy (0.44 points better) in comparison to the previously used Stanford parser. Furthermore, with the addition of psycholinguistic features, we achieve the strongest result to date reported on this task. Part of this work is published in Ambati et al. (2016c).

Chapter 7: Transition-based CCG Parsing using Neural Network Models This chapter presents a neural network based transition-based CCG parser, the first neural-network parser for CCG. We also study the impact of neural network based tagging models, and greedy versus beam-search parsing, by using a structured neural network model. We experiment with both English and Hindi CCGbanks. For English, our greedy parser obtains a labelled F-score of 83.27%, the best reported result for greedy CCG parsing in the literature (an

improvement of 2.5% over a perceptron based greedy parser) and is more than three times faster. For Hindi, our greedy parser achieves a labelled F-score of 74.14% which is an improvement of 3% over the greedy perceptron parser. In case of beam, structured neural network model, though not the state-of-the-art, consistently gave better results than the basic neural network model. Part of this work is published in Ambati et al. (2016b).

Chapter 8: Conclusion We conclude with possible future directions of this thesis in this chapter.

Chapter 2

Hindi CCGbank from Dependency Treebank

In this chapter, we first present an approach for automatically creating a Combinatory Categorial Grammar (CCG) treebank from a dependency treebank for the Subject-Object-Verb language Hindi. Rather than a direct conversion from dependency trees to CCG trees, we propose a two stage approach: a language independent generic algorithm first extracts a CCG lexicon from the dependency treebank. A deterministic CCG parser then creates a treebank of CCG derivations. We also discuss special cases of this generic algorithm to handle linguistic phenomena specific to Hindi. In doing so we extract different constructions with long-range dependencies like coordinate constructions and non-projective dependencies resulting from constructions like relative clauses, noun elaboration and verbal modifiers. Content of most of this chapter is published in Ambati et al. (2013) and Ambati et al. (2016a).

2.1 Introduction

Creation of CCGbanks in other languages, especially languages typologically far from English is beneficial both for the development of CCG analyses for linguistic phenomena in these languages, and also for the development of deep NLP tools for these languages. Different grammar formalisms like phrase structure grammar, combinatory categorial grammar, and dependency grammar have different advantages. But developing treebanks manually in each formalism is a very expensive and time consuming task. Automatic conversion of treebanks from one formalism to another significantly reduces the manual annotation effort. We develop an algorithm for automatically cre-

ating CCGbanks from dependency treebanks. We apply this approach to automatically creating a Hindi CCGbank from an existing manually created Hindi dependency treebank (Bhatt et al., 2009). The approach is applicable for creating CCGbanks for other languages with existing dependency treebanks, and is especially relevant for other Indian languages.

As compared to English, many Indian languages, including Hindi, while basically verb final, have a freer word order and are morphologically richer. All of these characteristics pose challenges to statistical parsers. In the Hindi dependency treebank there are around 20% of dependency trees with at least one non-projective arc which are problematic for vanilla shift-reduce parsing algorithms like arc-eager and arc-standard(Nivre et al., 2007b). In this work, we show that CCG can capture these phenomena elegantly, essentially by making such dependencies projective – that is, covered by the grammar. Our approach can be adapted to extract CCGbanks for other typologically similar languages with existing dependency treebanks, such as other Indic languages.

In this chapter we first present the related work regarding the automatic creation of CCGbanks for English and other languages (Section 2.2). A brief summary of the Hindi dependency treebank is provided in section 2.3. In sections 2.4 and 2.5, we first show how we extract a CCG lexicon from the Hindi dependency treebank and then use it to create a Hindi CCGbank. Details of different long-range dependencies arising from coordination and other non-projective constructions are presented in sections 2.6 and 2.7. Finally, an analysis of CCG categories and combinators present in the Hindi CCGbank is provided in section 2.8. We conclude with possible future directions in section 2.9.

2.2 Related Work

Hockenmaier and Steedman (2007) developed the first English CCGbank semi-automatically from the Penn Wall Street Journal Phrase Structure Treebank (Marcus et al., 1993). For each phrase structure tree, they first determine the constituent type of each node using heuristics adapted from Magerman (1994) and Collins (1999), which take the label of a node and its parent into account. Then the tree is binarized inserting dummy nodes as required into the tree such that all children to the left of the head branch off in a right-branching tree, and then all children to the right of the head branch off in a left-branching tree. Then CCG categories are assigned based on

whether the node is root of the sentence, complement or adjunct of the head. Finally, headword dependencies which approximate the underlying predicate-argument structure are obtained.

The English CCGbank (Hockenmaier and Steedman, 2007) is primarily created from the Penn Phrase Structure Treebank, which doesn't directly capture interesting linguistic phenomena like predicate-argument structures. Resources like PropBank (Palmer et al., 2005) capture predicate-argument structure of the verb. Using PropBank, Honnibal and Curran (2007) improved the complement and adjunct distinction in the CCGbank. Using information from different resources like PropBank and NomBank (Meyers et al., 2004), Honnibal et al. (2010) created an updated version of CCGbank which includes predicate-argument structures for both verbs and nouns, baseNP brackets, verb-particle constructions, and nominal modifiers. They also trained a state-of-the-art CCG parser on this new treebank and compared with the original treebank. Since the updated treebank contains fine-grained details the performance of the parser was slightly lower than the one trained on the original version.

Following Hockenmaier and Steedman (2007), there have been some efforts at automatically extracting treebanks of CCG derivations for other languages. Hockenmaier (2006) developed a CCGbank for German from the Tiger treebank (Brants et al., 2002). The Tiger treebank is based on a framework which has features from both phrase structure grammar and dependency grammar and results in graphs rather than trees. First, these graphs are pre-processed and converted to planar trees. Then a translation step is applied which binarizes the planar tree and extracts the CCG derivation. Tse and Curran (2010) use an algorithm similar to Hockenmaier and Steedman (2007) and extracted a Chinese CCGbank from the Penn Chinese Treebank (Xue et al., 2005).

There have also been work on extracting CCG lexicons (Cakici, 2005) and CCGbanks (Bos et al., 2009; Uematsu et al., 2013, 2015) from dependency treebanks. Bos et al. (2009) created an Italian CCGbank from the Turin University Treebank (TUT) [1], an Italian dependency treebank. They first converted dependency trees into phrase structure trees and then applying an algorithm similar to Hockenmaier and Steedman (2007) extracted the CCG derivations. Using different dependency resources available for Japanese like the Kyoto corpus (Kawahara et al., 2002) and the NAIST text corpus (Iida et al., 2007), Uematsu et al. (2013) developed a CCGbank for Japanese. They first integrated the dependency resources into phrase structure trees and then converted them into CCG derivations.

[1] http://www.di.unito.it/~tutreeb/

Cakici (2005) extracted a CCG lexicon for Turkish. She first made a list of complement and adjunct dependency labels. Traversing the dependency tree, she assigned CCG categories to each node based on complement or adjunct information. Following Cakici (2005), we first extract a Hindi CCG lexicon from the dependency treebank. Then we use a CKY parser based on the CCG formalism to automatically obtain a treebank of CCG derivations from this lexicon, a novel methodology that may be applicable to obtaining CCG treebanks in other languages as well. Our algorithm for extracting the lexicon is similar to Cakici (2005), but with pre-processing steps specific to Hindi. However, where Cakici (2005) extracted only a CCG lexicon, we extended it by developing a novel methodology for creating CCG derivations from this lexicon. Kumari and Rao (2015) have successfully applied our method to create a CCGbank for Telugu, an Indian language, differing from Hindi in belonging to the Dravidian language family, and being agglutinative. This shows that our algorithm is generic enough to be applied to other languages with little effort.

2.3 Hindi Dependency Treebank

In this section, we first give a brief introduction to the Hindi language. Then we provide details about the Paninian grammatical model used for Hindi dependency annotation. Following this, we describe the Hindi dependency treebank.

2.3.1 Hindi Language

Hindi is one of the official languages of the Republic of India, and the 4th largest language in the world, with over 260 million speakers[2]. Hindi, while basically verb final, is a freer word order language. This can be seen in (1), where (1a) shows the constituents in the default SOV (Subject, Object, Verb) order, and the remaining examples show some of the word order variants of (1a) [3].

(1) a. mohan ne raam ko kitaab dii.
Mohan ERG Ram DAT book give-past-fem
"Mohan gave a book for Ram" (S-IO-DO-V)

b. [mohan ne] [kitaab] [raam ko] [dii] (S-DO-IO-V)

c. [raam ko] [mohan ne] [kitaab] [dii] (IO-S-DO-V)

[2] http://www.ethnologue.com/statistics/size

[3] S=Subject; IO=Indirect Object; DO=Direct Object; V=Verb; ERG=Ergative; DAT=Dative

d. [raam ko] [kitaab] [mohan ne] [dii] (IO-DO-S-V)
e. [kitaab] [mohan ne] [raam ko] [dii] (DO-S-IO-V)
f. [kitaab] [raam ko] [mohan ne] [dii] (DO-IO-S-V)

Hindi also has a rich case marking system, although case marking is not obligatory. For example, in (1), while the subject and in-direct object are explicitly marked for the ergative [4] (ERG) and dative cases, the direct object is unmarked for the accusative.

2.3.2 Paninian Grammatical Model

Indian Languages (ILs) including Hindi are morphologically rich and have a relatively flexible word order. For such languages syntactic subject-object are not able to explain the varied linguistic phenomena. In fact, there is a debate in the literature whether the notions 'subject' and 'object' can be defined at all for ILs (Mohanan, 1982). Behavioural properties are the only criteria based on which one can confidently identify grammatical functions in Hindi (Mohanan, 1994); it can be difficult to exploit such properties computationally. Marking semantic properties such as thematic role as dependency relation is also problematic. Thematic roles are abstract notions and will require higher semantic features which are difficult to formulate and to extract as well. Paninian grammatical model (Kiparsky and Staal, 1969; Shastri, 1973) provides a level which while being syntactically grounded also helps in capturing semantics. In this section we briefly discuss the Paninian grammatical model for ILs and lay down some basic concepts inherent to this framework.

The Paninian framework considers information as central to the study of language. When a writer/speaker uses language to convey some information to the reader/hearer, he/she codes the information in the language string. Similarly, when a reader/hearer receives a language string, he/she extracts the information coded in it. The Paninian grammatical model is primarily concerned with: (a) how the information is coded and (b) how it can be extracted.

Two levels of representation can be readily understood in language: One, the actual language string (or sentence), two, what the speaker has in his mind. The latter can also be called as the meaning. Paninian framework has two other important levels: karaka level and vibhakti level

[4] Hindi is split-ergative. The ergative marker appears on the subject of a transitive verb with perfect morphology.

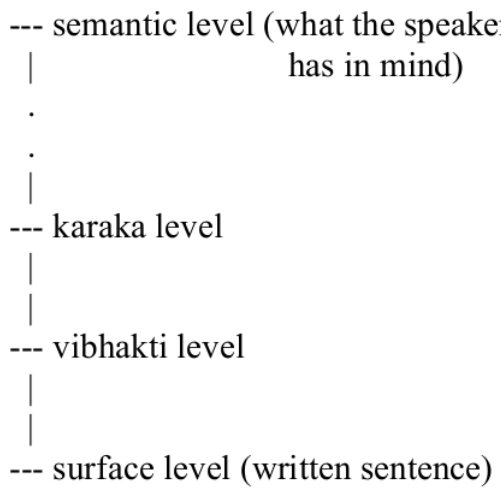


Figure 2.1: Levels of representation/analysis in the Paninian model

The surface level is the uttered or the written sentence. The vibhakti level is the level at which there are local word groups together with case endings, preposition or postposition markers. The vibhakti level abstracts away from many minor (including orthographic and idiosyncratic) differences among languages. Above the vibhakti level is the karaka level. It includes karaka relations and a few additional relations such as purpose. The topmost level relates to what the speaker has in his mind. This may be considered to be the ultimate meaning level that the speaker wants to convey. One can imagine several levels between the karaka and the ultimate level, each containing more semantic information. Thus, karaka level is one in a series of levels, but one which has relationship to semantics on the one hand and syntax on the other. The levels of representation in the Paninian model are presented in Figure 2.1.

At the karaka level, we have karaka relations and verb-verb relations, etc. Karaka relations are syntactico-semantic relations between the verbs and other related constituents (typically nouns) in a sentence. They capture a certain level of semantics which is somewhat similar to thematic relations but different from it (Bharati et al., 1995). This is the level of semantics that is important syntactically and is reflected in the surface form of the sentence(s). Begum et al. (2008b) have subsequently proposed an annotation scheme based on Paninian framework. They have extended the original formulation to account for previously unhandled syntactic phenomenon.

The Paninian approach treats a sentence as a set of modifier-modified relations. A sentence is supposed to have a primary modifier which is generally the main verb of the sentence. The elements modifying the verb participate in the action specified by the

verb. The participant relations with the verb are called karaka. The notion of karaka will incorporate the 'local' semantics of the verb in a sentence, while also taking cue from the surface level morpho-syntactic information (Vaidya et al., 2009). There are six basic karakas, namely;

- k1: karta (This is similar to subject or agent): the most independent participant in the action
- k2: karma (roughly the theme or object): the one most desired by the karta
- k3: karana (instrument): which is most essential for the action to take place
- k4: sampradaan (beneficiary): recipient or beneficiary of the action
- k5: apaadaan (source): movement away or separation from a source
- k7: adhikarana (location): location of the action in time and space

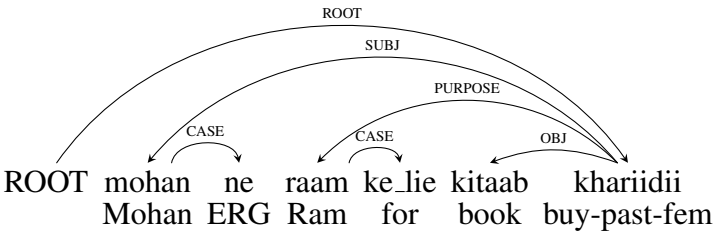
From the above description, it is easy to see that this analysis is a dependency based analysis (Kiparsky and Staal, 1969; Shastri, 1973), with verb as the root of the tree along with its argument structure as its children. The labels on the edges between a child-parent pair show the relationship between them. In addition to the above six labels many other have been proposed as part of the overall framework (Begum et al., 2008b; Bharati et al., 2009b). Appendix A.1 shows the most frequent dependency labels with their English equivalent.

In the following section, we provide details of the treebank annotated for Hindi using this Paninian grammatical model.

2.3.3 Treebank

In this work, we consider a subset of the Hindi Dependency Treebank (HDT ver-0.5) released as part of Coling 2012 Shared Task on parsing (Bharati et al., 2012). HDT is a multi-layered dependency treebank (Bhatt et al., 2009) annotated with morpho-syntactic (morphological, part-of-speech and chunk information) and syntactico-semantic (dependency) information (Bharati et al., 2006, 2009b). POS and chunk information is annotated following the POS and chunk annotation guidelines (Bharati et al., 2006). The morphological features have eight mandatory feature attributes for each node. These features are classified as root, coarse POS category, gender, number, person, case, post position (for a noun) or tense aspect modality

(for a verb) and suffix. The dependency annotation follows the Paninian grammar scheme described in section 2.3.2 which is known to be well-suited to modern Indian languages. Dependency labels are fine-grained, and mark dependencies that are syntactico-semantic in nature, such as agent (usually corresponding to subject), patient (object), and time and place expressions. There are special labels to mark long distance relations like relative clauses, coordination etc (Bharati et al., 1995, 2009b). Figure 2.2 presents the dependency tree for an example sentence 'mohan ne raam ke lie kitaab khariidi (Mohan bought a book for Ram)'.



'Mohan bought a book for Ram.'

Figure 2.2: An example dependency tree for Hindi (ERG = Ergative case).

In this example, the verb *khariidii* ("bought") is the root of the sentence. *mohan* ("Mohan") is the subject (SUBJ) of the verb *khariidii* ("bought") and *kitaab* ("book") is the object (OBJ) of the verb. Since the book is bought for *raam* ("Ram"), *raam* is attached to the verb with PURPOSE dependency label. The post-position markers *ne* (Ergative case marker) and *ke_lie* (equivalent to preposition "for") are attached to corresponding nouns with CASE dependency label.

The Hindi dependency treebank contains 12,041 training, 1,233 development and 1,828 testing sentences with an average of 22 words per sentence. Data is provided in the Shakti Standard Format (Bharati et al., 2007) and CoNLL format. The CoNLL format contains word, lemma, pos-tag, and coarse pos-tag in the WORD, LEMMA, POS, and CPOS fields respectively and morphological features, and chunk information in the FEATS column.[5] We use CoNLL format for all our experiments.

[5] http://nextens.uvt.nl/depparse-wiki/DataFormat

2.4 Extracting a CCG Lexicon

We first make a list of argument and adjunct dependency labels in the treebank. We obtained this list from the Hindi verb frames which make a distinction between arguments and adjuncts for different verbs Begum et al. (2008a). For example, dependencies with the label `k1` and `k2` (corresponding to subject and object respectively) are considered to be arguments, while labels like `k7p` and `k7t` (corresponding to place and time expressions) are considered to be adjuncts. For readability reasons, we will henceforth refer to dependency labels with their English equivalents (e.g., SUBJ, OBJ, PURPOSE, CASE for `k1`, `k2`, `rt`, `lwg__psp` respectively). A list of the Hindi dependency labels and their English equivalents are provided in the Appendix A.1.

Starting from the root of the dependency tree, we traverse each node. The category of a node depends on both its parent and children. If the node is an argument of its parent, we assign the chunk tag of the node (e.g., NP, PP) as its CCG category. Otherwise, we assign it a category of $X|X$, where X is the parent's *result* category and $|$ is directionality ($\backslash$ or $/$), which depends on the position of the node w.r.t. its parent. The *result* category of a node is the category obtained once its argument slots are saturated. For example, S_f, is the result category for $(S_f\backslash NP)\backslash NP$. Once we get the partial category of a node based on the node's parent information, we traverse through the children of the node. If a child is an argument, we add that child's chunk tag, with appropriate directionality, to the node's category. If the child is an adjunct, the category of the node is not affected.

Consider the verb *khariidii* ("bought") in the example sentence in Figure 2.3. Since it is the root of the sentence which is an argument dependency label, it gets a category S_f, from its parent. It has three children *mohan* ("Mohan"), *raam* ("Ram") and *kitaab* ("book"). We traverse through each child and update the category of *khariidii* as follows. *Mohan* is subject ("`SUBJ`") of *khariidii*. Since `SUBJ` is a mandatory argument, the category of *khariidii* is updated to $S_f\backslash NP$. The dependency label between *raam* and *khariidii* is `PURPOSE` which is an adjunct label. So, the category of *khariidii* ("bought") is not changed due to this child. The third and final child *kitaab* is an object ("`OBJ`") of the verb, which is an argument label. As a result, the category of *khariidii* is updated to $(S_f\backslash NP)\backslash NP$.[6]

Now we consider again the children of the verb *khariidii* ("bought"). *mohan* ("Mohan") is an argument of *khariidii*, and hence *NP* is the category for this node. *mohan*

[6]We return below to the question of case marking and agreement.

("Mohan") has a case marker *ne* ("ERG") as a child with dependency label CASE. Since CASE is an NP adjunct [7], the category of *mohan* ("Mohan") is not changed and remains *NP*. Now consider the child of *mohan* ("Mohan") which is *ne* ("ERG"). It is an adjunct that is to the right of its parent. Since *NP* is the result category of its parent *mohan* ("Mohan"), category of *ne* ("ERG") will be $NP\backslash NP$. Categories of other nodes are assigned similarly.

The algorithm is sketched in Figure 2.4 and an example of a CCG derivation for a simple sentence, marked with chunk tags, is shown in Figure 2.3. NP and S_f are the chunk tags for noun and finite verb chunks respectively[8]. Some important special cases are described in detail in the following subsections.

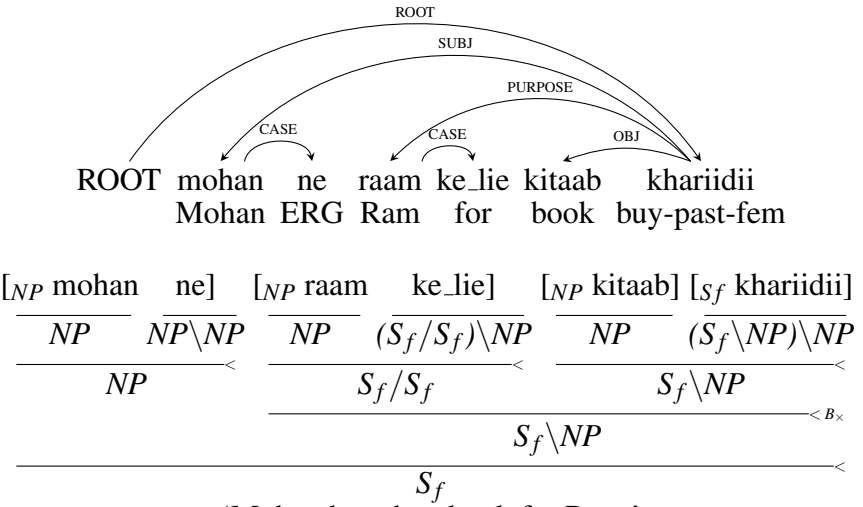


Figure 2.3: An example dependency tree with its CCG derivation.

The process described above yields a "coarse-grained" lexicon, in which case is not distinguished. We also created a "fine-grained" lexicon, in which we retain morphological information in noun categories. For example, consider the noun chunk *raam ne* ("Ram ERG"). In the fine-grained lexicon, the CCG categories for *raam* and *ne* are *NP* and *NP[ne]\NP* respectively. Morphological information such as ergative case '-ne' in noun categories is expected to help with determining their dependency labels, but makes the lexicon more sparse. We therefore extract both a coarse-grained and a fine-grained lexicon; details of the machine readable format for both lexicons is presented

[7] We treated CASE as NP adjunct for the case of consistency with the dependency treebank. We leave the other ways of treating CASE for future work.

[8] VGF is the chunk tag for finite verb chunk in the Hindi dependency treebank. But for the sake of brevity we use S_f notation here. A list of the Hindi chunk tags are provided in the Appendix A.1.

```
ModifyTree(DependencyTree tree);
for (each node in tree):
   handlePostPositionMarkers(node);
   handleSpecialCases(node);
   if (node is an argument of parent):
     cat = node.chunkTag;
   else:
     prescat = parent.resultCategory;
     cat = prescat + getDir(node, parent) + prescat;
  for(each child of node):
     if (child is an argument of node):
        cat = cat + getDir(child, node) + child.chunkTag;
```

Figure 2.4: Algorithm for extracting a CCG lexicon from a dependency tree.

in Appendix A.2.

2.4.1 Morphological Markers

In Hindi, morphological information is encoded in the form of post-positional markers on nouns, and tense, aspect and modality markers on verbs. A post-positional marker following a noun plays the role of a case-marker (e.g., *raam ne* (“Ram ERG”), here *ne* is the ergative case marker) and a role similar to an English preposition (e.g., *mej par* (“table on”), here *par* is the postpositional equivalent of the English preposition “on”). Post-positional markers on nouns can be simple one word expressions like *ne* or *par*, or multiple words as in *raam ke lie* (“Ram for”). Complex post position markers as a whole give information about how the head noun or verb behaves. For example, *ke lie* is equivalent to “for” and *ke baare me* is equivalent to “about”. The Hindi CCGbank merges complex postpositional markers into single words like *ke_lie* so that the entire marker gets a single CCG category.

For the “fine-grained” lexicon, we explored two variants of the lexicon: normal and type-raised. In the normal version, the ergative case marker like *ne* looks for an *NP* to the left and forms the CCG category *NP[ne]*. Whereas in the type-raised version, the category of *ne* takes an *NP* to its left and creates a category which looks for an

intransitive verb $S_f\backslash NP[ne]$.

$$\dfrac{\dfrac{\begin{array}{c}\text{raam}\\ \text{Ram}\end{array}}{NP} \quad \dfrac{\begin{array}{c}\text{ne}\\ \text{ERG}\end{array}}{(S/(S_f\backslash NP[ne]))\backslash NP}}{S/(S_f\backslash NP[ne])}<$$

For an adjunct like *raam ke_lie* ("for Ram") in Figure 2.3, we pass the adjunct information to the post-position marker *ke_lie*, with *NP* as the category for the head noun phrase, and the category $(S_f/S_f)\backslash NP$ for the postposition. Adjuncts that modify adjacent adjuncts are assigned identical categories X/X making use of CCG's composition rule and following Cakici (2005).

2.5 CCG Lexicon to Treebank conversion

Phrase structure to CCG conversion algorithms like Hockenmaier and Steedman (2007) first convert a phrase structure tree into a binary tree. Converting a dependency tree into a binary tree is not possible in the presence of a non-projective arc. There are around 20% of sentences in the Hindi dependency treebank with at least one non-projective arc. We therefore use a CCG parser to convert the CCG lexicon to a CCG treebank, a novel idea, as conversion to CCG trees directly from dependency trees is not straight-forward due to the above reason.

Using the algorithm presented in the previous section, we obtained one CCG category for every word in a sentence. We then run a non-statistical CKY chart parser based on the CCG formalism[9], which gives CCG derivations based on the lexical categories. This gives multiple derivations for some sentences. We rank these derivations using two criteria. The first criterion is correct recovery of the gold dependency tree. Derivations which lead to gold dependencies are given higher weight. In the second criterion, we prefer derivations which yield intra-chunk dependencies (e.g., verb and auxiliary) prior to inter-chunk (e.g., verb and its arguments). For example, morphological markers (which lead to intra-chunk dependencies) play a crucial role in identifying correct dependencies. Resolving these dependencies first helps the parser in better identification of inter-chunk dependencies such as argument structure of the verb (Ambati, 2011). We thus extract the best derivation for each sentence and create a CCGbank for Hindi.

[9] http://openccg.sourceforge.net/

Coverage, i.e., number of sentences for which we got at least one complete derivation, using this lexicon is 96%. Disabling crossed composition reduced the coverage by around 10% showing the importance of this rule for a free word order language with 20% non-projective sentences. The remaining 4% are either cases of inconsistent annotations in the original treebank, or constructions which are currently not handled by our conversion algorithm.

We extracted dependencies from the CCG Treebank and evaluated them with the dependencies in the dependency treebank. Similar to Clark and Curran (2007), we use indexed categories to achieve this. For example, $(S\backslash NP_1)\backslash NP_2$ is the indexed category of $(S\backslash NP)\backslash NP$. NP_1 resolves the subject dependency and NP_2 resolves the object dependency. Hindi CCGbank captures 99.1% of the dependencies in the dependency treebank.

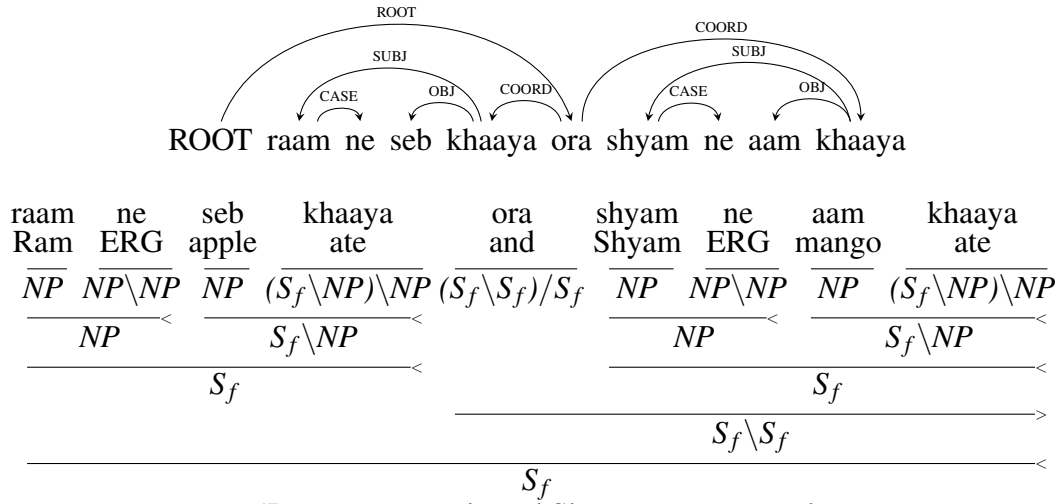
2.6 Coordination Constructions

Coordination is one of the most frequent sources of long distance dependencies in corpora. Coordination can occur between similar components like noun-noun coordination, verb-verb coordination or between compatible components like adjective-noun coordination. The CCG category of a conjunction is $(X\backslash X)/X$, where a conjunction looks for a child of type X to its right and then a child to its left of the same type X to yield a result of the same type X. Figure 2.5 gives dependency tree and CCG derivation for an example sentence with sentential (S) coordination. In the Hindi CCGbank, it is the supertagger that identifies the correct instantiation of the type X for the conjunction.[10]

There are four major types of coordination constructions in Hindi. We will describe each type with an example and explain how CCG handles them.

Type 1 (Conjunction with two children): The CCG category of the conjunction is $(X\backslash X)/X$ where X depends on the category of the conjuncts. The example given below in figure 2.6, *raam ora shyam skool gaye* (“Ram and Shyam went to school”), is the case of noun-phrase (NP) coordination. Conjunct *ora* (“and”) has two noun phrases *raam* (“Ram”) and *shyam* (“Shyam”) as its children. Hence the category of *ora* (“and”) is $(NP\backslash NP)/NP$. *ora* (“and”) is first combined with the right child *shyam* and then combined with the left child *raam* leading to a noun phrase, which becomes

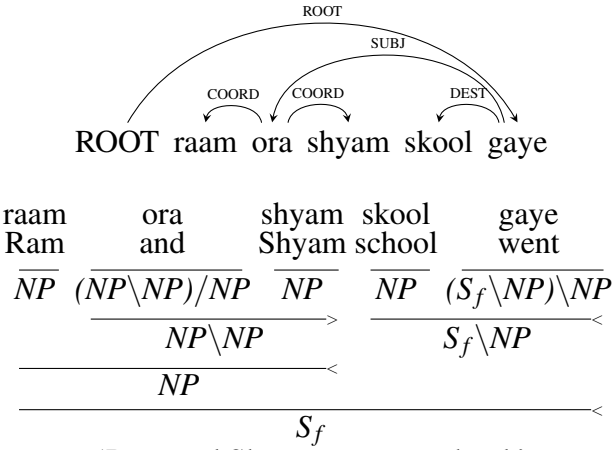
[10]This treatment constitutes a slight difference from English CCGbank, where coordination is treated syncategorematically, with conjunction bearing the category *conj*.



'Ram ate an apple and Shyam ate a mango.'

Figure 2.5: Sentential coordination.

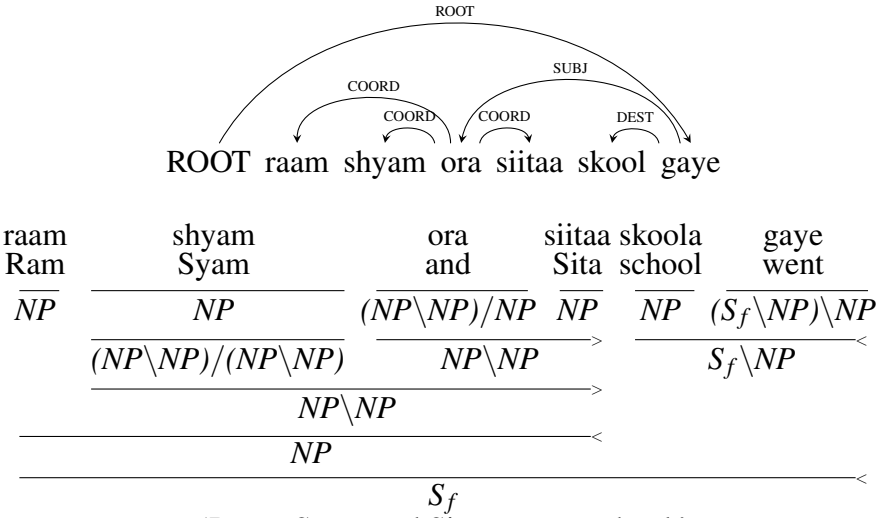
the subject argument for the verb *gaye* ("went").



'Ram and Shyam went to school.'

Figure 2.6: Type 1 coordination.

Type 2 (Conjunction with more than two children and not separated by commas): In Hindi, sometimes a conjunction can have more than two children which are not separated by commas. In such cases, CCG category of the node is type-changed from *X* to a category *(X\X)/(X\X)*. Figure 2.7 shows the dependency tree of an example sentence *raam shyam ora sita skool gaye* ("Ram Shyam and Sita went to school"). In this example, the conjunct *ora* ("and") has three children *raam* ("Ram"), *shyam* ("Shyam") and *sita* ("Sita"). CCG category of *shyam* is type-changed from *NP* to *(NP\NP)/(NP\NP)* so that it can combine with *ora* and then with *raam* to form an *NP*.



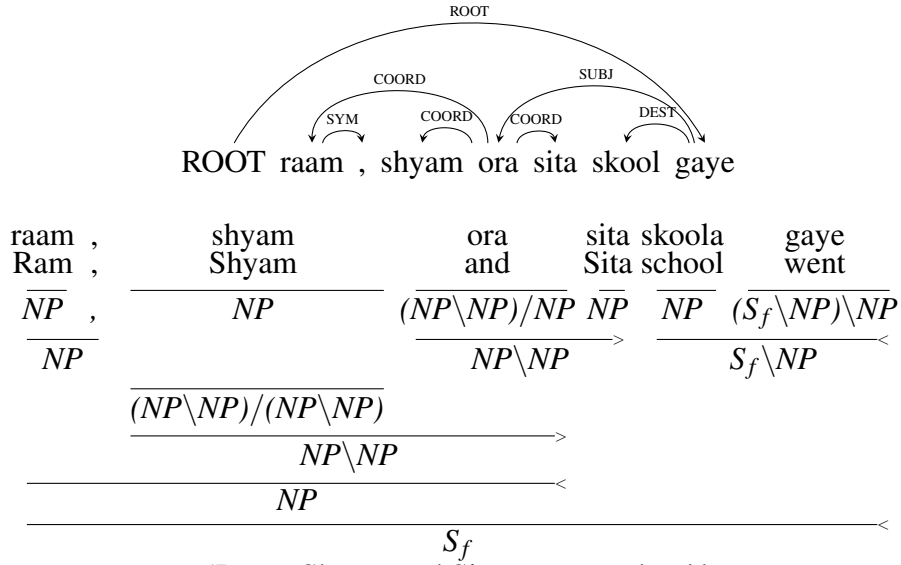
'Ram , Syam and Sita went to school.'

Figure 2.7: Type 2 coordination.

Type 3 (Conjunction with more than two children separated by commas): The example sentence given below in Figure 2.8, *raam , shyam ora sita skool gaye* ("Ram, Shyam and Sita went to school"), is the same as the one presented above in Type 2 category. The only difference is that there is a comma between the nouns *raam* ("Ram") and *shyam* ("Shyam"). The comma gets a CCG category , which is combined with *NP* to form an *NP*. Similar to Type 2, the CCG category of *shyam* is type-changed from *NP* to *(NP\NP)/(NP\NP)*. This allows *shyam* to combine with *ora* and then with *raam* to form an *NP*.

Unlike other CCGbanks which treat comma as a conjunction, we treat comma as a punctuation here. In that way, we don't have to change the dependency tree. If we treat a comma as a conjunction, then we have to change the dependency tree as well, where *ora* ("and") will have comma and *sita* as children and comma will have *raam* and *shyam* as children. Also, since comma can be missing as in Type 2, treating the comma as a punctuation leads to having a single analysis irrespective of whether a comma is present or not.

Type 4 (Argument cluster coordination): Figure 2.9 presents an example sentence for argument cluster coordination, *raam ne seb ora sita ne aam khaaya* ("Ram ate an apple and Sita ate a mango"). *khaaya* ("ate") is the shared verb for both the sentences. To handle such constructions, dependency tree introduces a dummy "NULL" node which is co-indexed with the main verb *khaaya* and acts as the verb for the 1st sentence as shown in the dependency tree in Figure 2.9.



'Ram , Shyam and Sita went to school.'

Figure 2.8: Type 3 coordination.

CCG can handle such constructions without introducing NULL nodes. The subject *raam ne* is type-raised from *NP* to a category which looks for an intransitive verb, $S_f/(S_f\backslash NP)$. Similarly, the object *seb* ("apple") is type-raised from *NP* to a category which looks for a transitive verb, $(S_f\backslash NP)/((S_f\backslash NP)\backslash NP)$. Now, these two nodes are combined leading to $S_f/((S_f\backslash NP)\backslash NP)$ which takes a transitive verb and forms a sentence. Similarly, subject and object arguments of the second sentence, *sita ne* ("Sita") and *aam* ("Mango") are type-raised and combined. Now, these type-raised arguments are combined using the conjunction *ora* ("and") which is then combined with the main verb *khaaya* to form a sentence [11].

2.7 "Non-Projective" Constructions

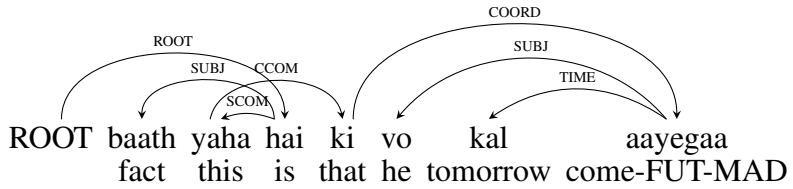
In the tradition of dependency grammar (Hays, 1964), constructions which induce dependency arcs which cross as in Figure 2.10 are referred to as "non-projective", because they cannot be generated by the core context-free dependency grammar, and are generally supposed to arise from some separate component of the grammar, such as transformational rules (Robinson, 1970).

Such dependencies arise in all languages from processes like relativization and

[11] We are not handling argument cluster coordination in the current version of the Hindi CCGbank since the current version doesn't include unary type-changing rules. We will handle these constructions in the next version.

‘Ram ate an apple and Sita ate a mango.’

Figure 2.9: Type 4 coordination.



'The fact is that he will come tomorrow'

Figure 2.10: A dependency tree with a "non-projective" dependency.

various instances of coordination reduction. To call them "non-projective" is confusing in the present context, since the central claim of CCG is that *all* dependencies are projective, in the sense of arising directly from near-context free syntactic projection. In the dependency parsing literature techniques like swap action (Nivre, 2009) or pseudo-projective parsing algorithm (Nivre and Nilsson, 2005) are used to handle these crossing arcs. In case of CCG, we can extract such crossing dependencies using indexed categories[12]. Section 2.7.3 provides an example derivation showing how indexed categories can be used to extract crossing dependencies. In this section, we present different constructions and/or dependency labels which lead to crossing arcs in the dependency treebank, and explain how CCG can be made to handle them projectively.

Because Hindi has a comparatively free word-order, crossing dependencies are more frequent in the Hindi dependency treebank than in comparable English data. There are a total of 20% sentences with non-projective arcs in the Hindi dependency treebank, amounting to 1.1% of total arcs. There is some previous work on analyzing different non-projective constructions in Hindi and other Indian languages (Mannem et al., 2009; Bhat and Sharma, 2012). We categorize the non-projective constructions in the Hindi dependency treebank based on this previous work. Table 2.1 shows the distribution of non-projective arcs across different constructions.

In the following sections, we present different constructions which lead to crossing arcs in the dependency treebank, and explain how CCG can be made to handle them projectively.

[12]Please refer to Clark and Curran (2007) for the details on how indexed categories are used to extract dependencies

Type of Construction	Percentage (%)
Clausal Complements	32.4
Relative Clause Constructions	19.7
Topicalization	15.3
Genitives and Dislocated/Discontinuous Genitives	12.8
Paired Connectives	10.5
Others	9.3

Table 2.1: Distribution of different non-projective constructions in the treebank.

2.7.1 Clausal Complements

Clausal complements forming a complex NP are the cases where clauses elaborate on a noun/pronoun. These are annotated with the CCOM dependency label. For example, in the sentence given in Figure 2.11, *baat* ("fact") is the subject ("SUBJ") and *yaha* ("this") is its noun complement ("SCOM"), which are attached to the verb. Whereas the clause *ki vo kal aayegaa* ("that he will come tomorrow") has a dependency relation with *yaha* ("this") and is denoted by CCOM dependency label. 32% of crossing arcs in the treebank are due to this construction.

There are two options to handle this case. In the first option we don't change the dependency tree. Since *ki* ("that") is a subordinate conjunction, its chunk tag is CCP. As it looks for a clause/sentence to its right, CCG category for *ki* ("that") will be CCP/S_f. This gives *yaha* ("this") a CCG category of NP/CCP, since the result category of its child *ki* ("that") is *CCP*. We can combine *yaha* ("this") and *hai* ("is") using Backward Crossing Composition ($< B_\times$) which can then be combined with *ki* ("that") to establish the crossing dependency. Figure 2.11 gives the CCG derivation for this example.

Another option is to systematically change the dependency trees concerned to make the complementizer *ki* ("that") the child of the copula *hai* ("is"), rather than of the demonstrative *yaha* ("this"). As a result, the complementizer *ki* is assigned the category $(S_f \backslash S_f)/S_f$, which will first combine with the clause to its right *vo kal aayegaa*, and then with the clause to its left *baat yaha hai*, resulting in the derivation shown below in Figure 2.12. For the CCGbank conversion, we modified the dependency tree since we can avoid the use of crossed composition and thus simplifying the conversion process without losing the linguistic information [13].

[13] It is easy to re-construct the original dependency with the help of lexical item *yaha* ("this"). We

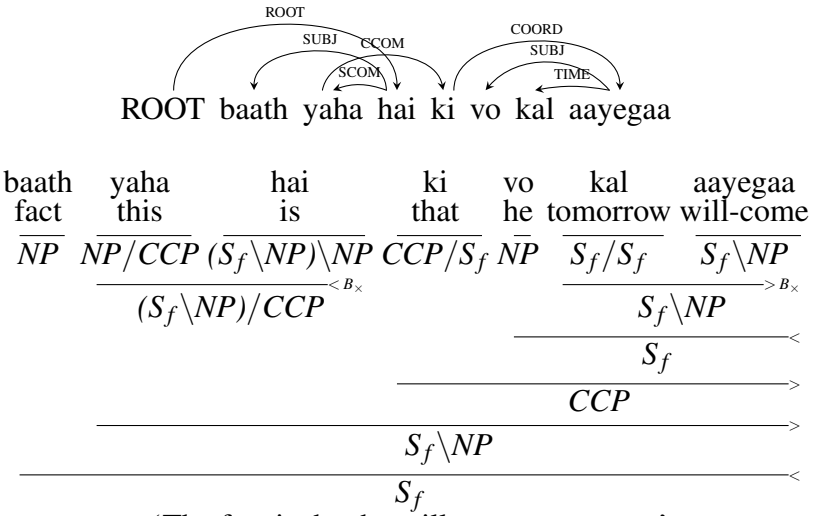


Figure 2.11: CCOM: CCG Derivation (Original dependency tree).

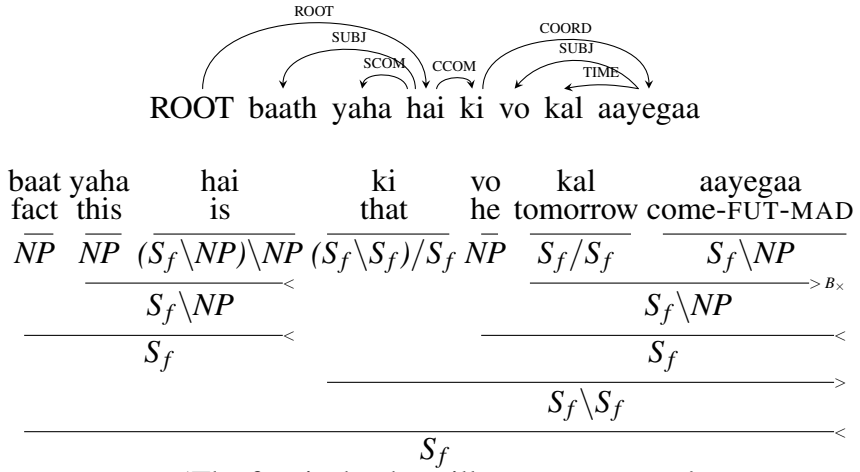


Figure 2.12: CCOM: CCG Derivation (Modified dependency tree).

2.7.2 Relative Clause Constructions

Relative clauses are the second major constructions which lead to crossing dependency arcs in the original treebank. 20% of such arcs in the data are due to relative clauses. In English, relative clauses have the category type $NP\backslash NP$, where they combine with a noun phrase on the left to give a resulting noun phrase. Hindi has relative clauses of the type NP\NP or NP/NP based on the position of the relative clause with respect to the head noun.

For instance, for the example sentence in Figure 2.13, the relative clause has

can find the parent of *ki* ("that") and extract the lexical item *yaha* ("this") from its sub-tree. Assigning it as the parent of *ki* ("that") would result in the original dependency tree.

$NP\backslash NP$ as its CCG category, since it is to the right of the head noun. Whereas in Figure 2.14, the category of the relative clause is NP/NP since it is to the left of the head noun. Similar to English, in Hindi also, we pass down this information to the relative pronoun rather than the main verb of the relative clause. As a result, the relative pronoun will have a CCG category of $(NP|NP)|X$ where the directionality depends on the position of the relative pronoun in the clause and the category X depends on the grammatical role of the relative pronoun.

Embedded: This is a simple case of relative clause where the relative clause is to the right of its head noun. Mahajan (2000) calls such constructions as "Normal" since it is similar to English relative clause construction. This type of relative clause doesn't lead to crossing dependency arcs. Figure 2.13 gives an example sentence, *vo ladakaa jo khadaa hai raam hai* ("The boy who is standing is Ram") with its dependency tree and corresponding CCG derivation [14]. The relative clause is marked within the brackets in the following figure. In this example, the category of the relative pronoun *jo* ("who") is $(NP\backslash NP)/(S_f\backslash NP)$ which is similar to English relative pronouns. The relative pronoun *jo* ("who") first combines with the verb phrase *khadaa hai* ("is standing") to form a relative clause with category $NP\backslash NP$. The relative clause then combines with its head noun phrase *vo ladakaa* ("that boy") which is then combined with the main verb phrase to form a sentence S_f.

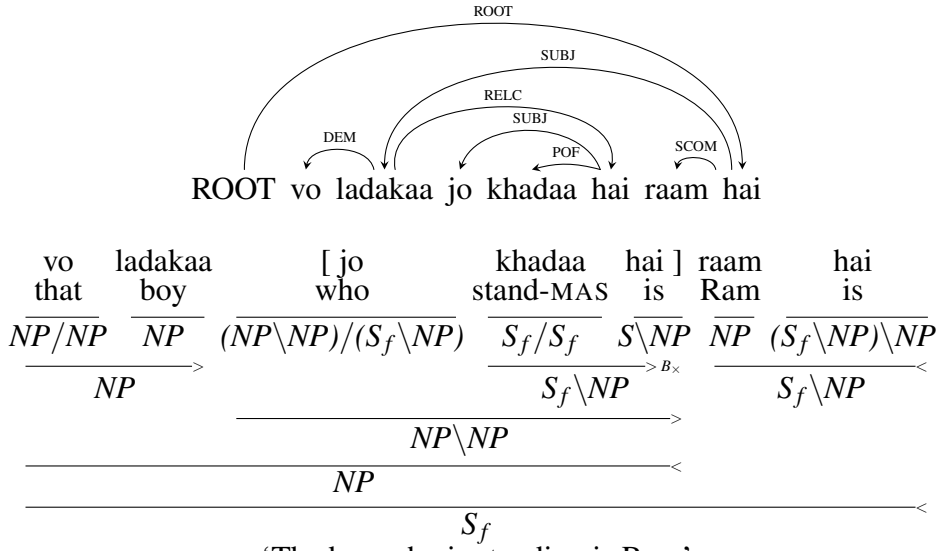


Figure 2.13: Embedded Relative Clause.

[14] In Hindi dependency treebank POF (part-of) dependency label is used to represent part of units such as conjunct verbs.

Correlatives: In Hindi, a relative clause can occur to the left of the head noun as well, which is the most frequent construction. This case of relative clause also doesn't lead to crossing dependency arcs. Figure 2.14 gives the dependency tree and corresponding CCG derivation for an example sentence, *jo ladakaa khadaa hai vah raam hai* ("The boy who is standing is Ram"). In this example, the relative pronoun *jo* ("who") occurs as a demonstrative. So the category of *jo* ('who") is $((NP/NP)/(S_f\backslash NP))/NP$. The relative pronoun *jo* ("who") combines with its head noun *ladakaa* ("boy") which is then combined with the verb phrase leading to the category of relative clause NP/NP. Since the relative clause is to the left of the head noun, its category is NP/NP rather than $NP\backslash NP$ which we saw in the previous embedded relative clause.

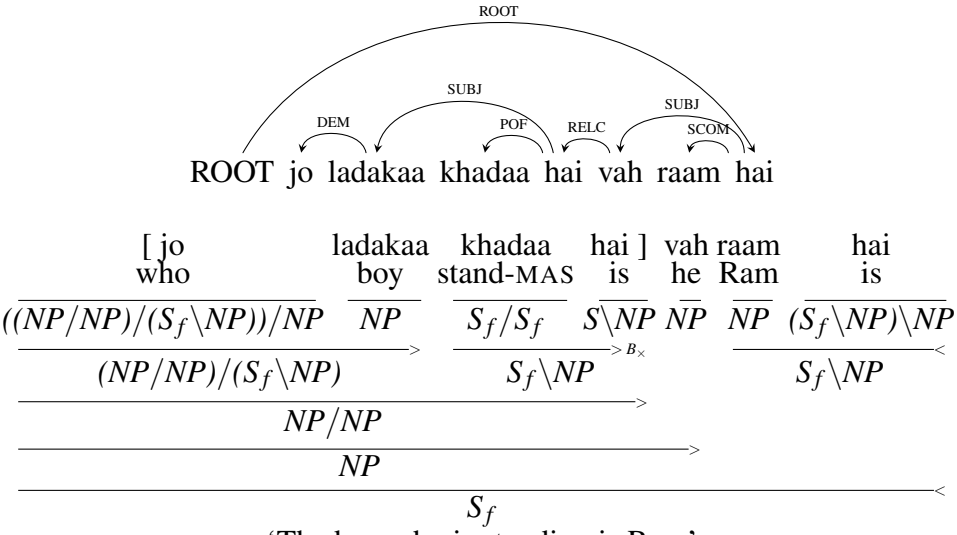


Figure 2.14: Correlatives.

Extraposed: Unlike the previous two cases of embedded and correlative constructions where the relative clause is next to the head noun, Hindi has constructions where the relative clause is not next to its head noun. Figure 2.15 shows one such example sentence *vah ladakaa raam hai jo khadaa hai* ("That boy is Ram who is standing"). This type of construction leads to a crossing dependency arc. To handle this we change the dependency tree slightly. Instead of the relative clause modifying the head noun, we make it modify the main verb. As a result the relative pronoun will have a CCG category of $(S|S)|X$ instead of $(NP|NP)|X$. Since this is a case of extraposed/dislocated relative clause, the category of relative clause is $S|S$ rather than $NP|NP$. The original dependency arc is marked with red dotted lines in Figure 2.15. Note that it is easy

to recover the dependency between the relative clause and its head noun, as the head noun chunk will have a word whose root is *vo* ("that")[15].

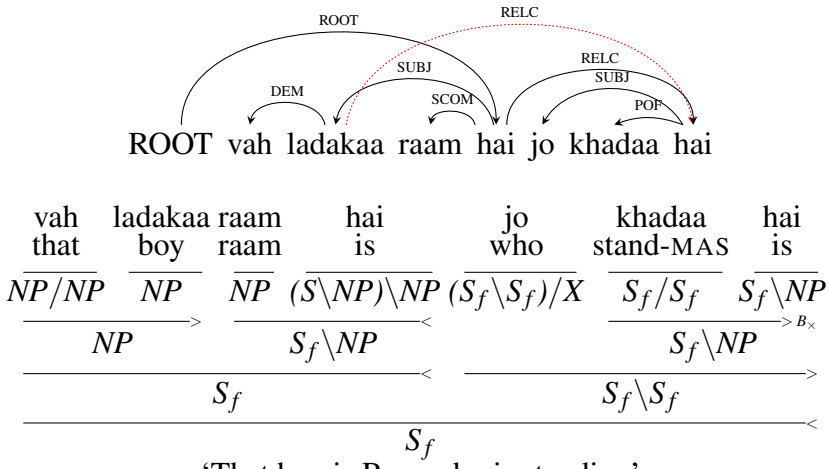


Figure 2.15: Extraposed Relative Clause (Example 1).

Figure 2.16 presents another example sentence which is similar to Figure 2.15, except that the relative pronoun is not at the starting of the relative clause and it is also not the mandatory argument of the verb of the relative clause. Here, the relative pronoun *jaisaa* ("like-what") is neither at the beginning of the clause nor a mandatory argument. It is an adverbial modifier (ADV) for the verb *kahaa* ("said"). As a result, the relative pronoun *jaisaa* will have a CCG category $(S_f/S_f)/S_f$. *jaisaa* is combined with the verb *kahaa* ("said") using forward crossed composition ($B_\times$) which leads to a category of S_f/S_f for the relative clause in the end. Similar to the previous example, this is a case of extraposed relative clause.

2.7.3 Topicalization

The node which is the object/patient of the verb is marked with OBJ dependency label. This OBJ label or topicalization is the cause for 11.3 % of crossing dependency arcs in the treebank.

Figure 2.17 presents an example sentence where a crossing arc is created due to the object (OBJ) relation. In the example sentence, *khaanaa raam khaakar dukaan gayaa*

[15]For example, in figure 2.15, CCG derivation gives the dependency between *hai* ("is") of relative clause and *hai* ("is") of main clause. As the chunk with *vo* ("that") root word (here *vaha*) is *vaha ladakaa* ("that boy"), the head of *hai* ("is") as per Hindi dependency guidelines would be *ladakaa* ("boy").

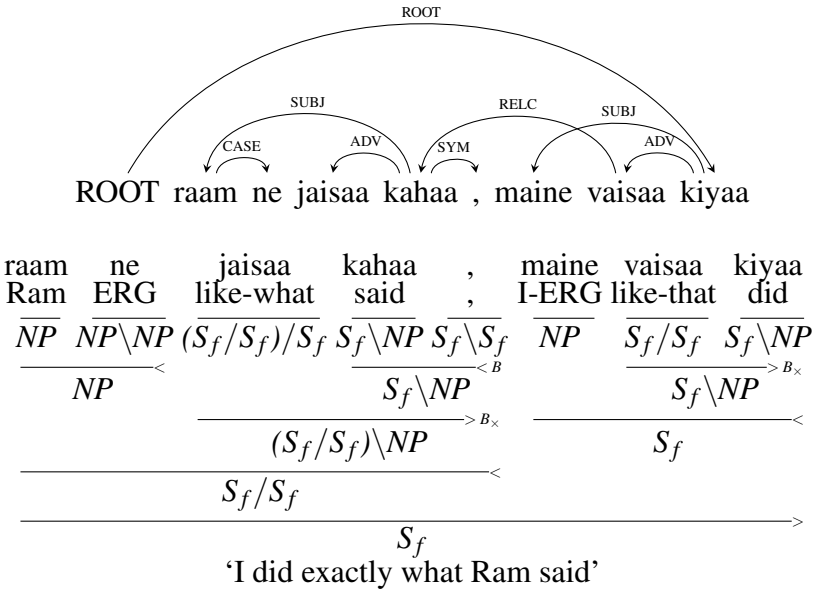
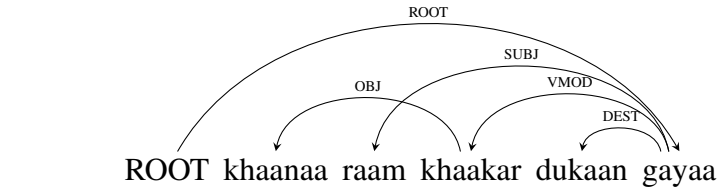


Figure 2.16: Extraposed Relative Clause (Example 2).

("Ram after eating food went to the shop"), there are two verbs: *khaakar* ("having-eaten"), a non-finite verb and *gayaa* ("went"), a finite verb. *raam* ("Ram") is the shared subject (SUBJ) of both the verbs. As per Hindi dependency guidelines, *raam* cannot have two parents. So it is marked as SUBJ of the main verb *gayaa* ("went"). If the subject, *raam*, was at the start of the sentence then the sentence would be *raam khaanaa khaakar dukaan gayaa*, which is the most frequent construction. Then it would not have created the crossing arc. Shared subject *raam* appearing within non-finite verb phrase *khaanaa khaakar* ("having eaten food") is not very common.

To handle these types of constructions, we relax the constraint of a node having multiple parents. *raam* is subject of both the verbs: *khaakar* ("having eaten") and *gayaa* ("went"). But, due to tree constraint, dependency tree avoids the subject *raam* having two parents. We let the CCG derivation have *raam* as the subject for both the verbs. As a result, *khaakar* ("having-eaten") will have the CCG category $((S_f/(S_f\backslash NP_2))\backslash NP_1)\backslash NP_2$[16]. The first part of the category, $(S_f/(S_f\backslash NP_2))$, captures the information that it is a verbal modifier which shares an argument with the main verb. *khaakar* ("having-eaten") first combines with *raam* and then with *khaanaa* ("food") to form $S_f/(S_f\backslash NP_2)$. This is then combined with the VP *dukaan gayaa* ("went to shop") resulting in a sentence S_f. Note that *gayaa* and *raam* are never combined directly in the derivation. But this dependency is resolved using the indices.

[16]Indices for categories are not part of the lexicon but indices are used while extracting dependencies from the CCG derivation.



khaanaa	raam	khaakar	dukaan	gayaa
food	Ram	having-eaten	shop	went
NP_1	NP_2	$((S_f/(S_f\backslash NP_2))\backslash NP_1)\backslash NP_2$	NP_3	$(S_f\backslash NP_2)\backslash NP_3$
		$(S_f/(S_f\backslash NP_2))\backslash NP_1$ (<)		$S_f\backslash NP_2$ (<)
		$S_f/(S_f\backslash NP_2)$ (<)		
		S_f (>)		

'Ram after eating food went to the shop'

Figure 2.17: Topicalization.

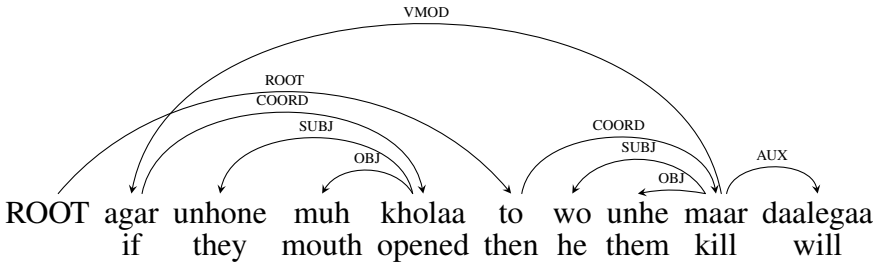
2.7.4 Paired Connectives

Paired connectives such as *agar-to* ("if-then") are the cause for 10.5% of crossing dependency arcs in the treebank. These constructions involve VMOD, verbal modifier, dependency label. Any verbal modifier which cannot be categorised as a specific relation like subject (SUBJ), object (OBJ) etc. is marked by a VMOD relation.

Original Annotation: Figure 2.18 presents an example 'if-then' construction. In the original dependency tree for this sentence, *agar unhone muh kholaa to wo unhe maar daalegaa* ("If they opened their mouth then he will kill them"), *to* ("then") is the ROOT of the sentence. *maar* ("kill") is the child of *tho* ("then") with the dependency relation COORD. *agar* ("if") is the child of *maar* ("kill") with dependency relation VMOD and *kholaa* ("opened") is the child of *agar* ("if") with dependency relation COORD. VMOD relation between *maar* ("kill") and *agar* ("if") leads to a crossing dependency arc here.

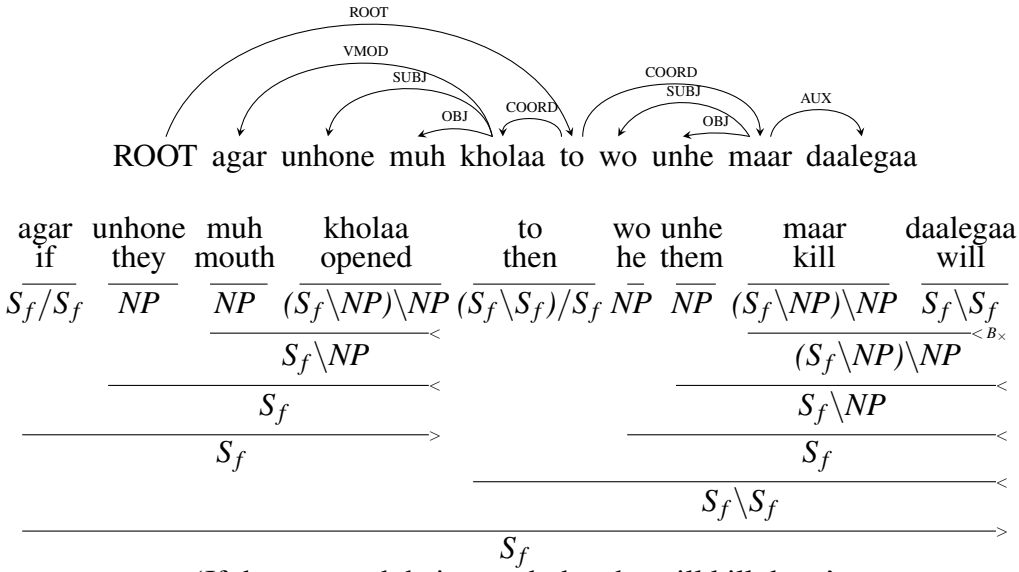
Modified Annotation: We modified the dependency tree to handle this construction. In the modified tree, *to* ("then") is still the ROOT of the sentence. Both the verbs *maar* ("kill") and *kholaa* ("opened") are children of *to* ("then") with a COORD dependency relation. *agara* ("if") is the child of *kholaa* ("opens") with the dependency relation VMOD.

In the case of English if-then constructions, the CCG category of *if* is *(S/S)/S[dcl]* which consumes a sentence to its right, leading to an S/S category for the if-clause. It then consumes the then-clause leading to S category. But in the case of Hindi *agar*



'If they opened their mouth then he will kill them'

Figure 2.18: Paired Connectives: Original dependency tree.



'If they opened their mouth then he will kill them'

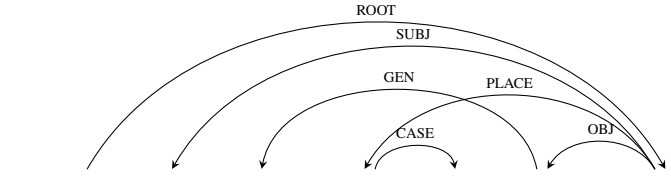
Figure 2.19: Paired Connectives: Modified dependency tree and corresponding CCG derivation.

("if") can be optional. To capture this phenomenon, we make the category of *tho* ("then") to demand *agar* ("if") clause rather than the opposite. So, the CCG category of *to* ("then") is $(S_f\backslash S_f)/S_f$ which consumes a sentence to its right forming a then-clause with the category $S_f\backslash S_f$. It then combines with a sentence to its left which is the if-clause leading to S_f. Also, as *agar* ("if") is optional it takes an adjunct category making the main verb the head of the clause. Figure 2.19 shows the modified dependency tree with the corresponding CCG derivation.

2.7.5 Genitives and Dislocated/Discontinuous Genitives

The genitive/possessive relation which holds between two nouns is marked by GEN dependency label. It mostly occurs with 'kaa' (masc.) or 'kii' (fem.) postposition marker. A reliable cue for its identification is that the postposition agrees with the noun it modifies in number and gender. In the majority of cases the nouns in genitive relation are next to each other. But, in some cases, due to the free word order nature of Hindi, some other word can occur between the two nouns in a genitive relation as in the following example in Figure 2.20. This construction is the source of 7.5% of the crossing arcs in the the dependency treebank.

In the example in Figure 2.20, *maine uskaa mumbai mai kiraayaa dediyaa* ("I have given his rent in Mumbai"), *uskaa* ("his") and *kiraayaa* ("rent") are in genitive relation. But, *mumbai mai* ("in Mumbai") is between these two nouns leading to a crossing arc. Though the dependency labels are different, the construction is similar to the ones described in Section 2.7.5. When two nouns are in a genitive relation, if the both the nouns are next to each other we make the noun with genitive marker demand a noun to its right similar to genitive cases in other languages. But, if both the nouns in genitive relation are not next to each other, then we make the head noun demand the noun with genitive marker as in Figure 2.20. In this way, we can capture this unusual word ordering elegantly in CCG.



maine	uskaa	mumbai	mai	kiraayaa	dediyaa
I-ERG	his	Mumbai	in	rent	have-given
NP	NP_{gen}	NP	$(S_f/S_f)\backslash NP$	$NP\backslash NP_{gen}$	$(S_f\backslash NP)\backslash NP$
		S_f/S_f (<)		$(S_f\backslash NP)\backslash NP_{gen}$ (<B)	
		$(S_f\backslash NP)\backslash NP_{gen}$ (>B×)			
	$S_f\backslash NP$ (<)				
S_f (<)					

'I have given his rent in Mumbai'

Figure 2.20: Genitive construction.

Hindi also has extensive use of conjunct "light" verbs. A conjunct verb is com-

posed of a noun or an adjective followed by a verbalizer. Subject (SUBJ) or Object (OBJ) arguments of a conjunct verb can have the genitive case marker. In such cases, the arguments have a dependency relation with the noun of the conjunct verb since the agreement is with the noun of the conjunct verb and not with the verb. The free word order nature of adverbs and time and/or place expressions can cause crossing arcs as in the following examples. Such constructions are called dislocated/discontinuous genitives. We treat Part-OF (POF) and subject/object of conjunct verb (CSUBJ/COBJ) as arguments. For example, in Figure 2.21, the light verb *hua* ("happened") looks for an *NP*, *udhghaatana* ("inauguration") to its left. *udhghaatana* has a child *mandir kaa* ("of temple") with CSUBJ dependency relation. Since CSUBJ is an argument relation, CCG category of *udhghaatana* is $NP\backslash NP_{gen}$ which looks for an NP with genitive marker to its left. *udhghaatana* first combines with the light verb *hua* and then with the optional time expression *kala* ("yesterday") leading to $S_f\backslash NP_{gen}$. The verb phrase $S_f\backslash NP_{gen}$ is then combined with the noun phrase with genitive marker *mandir kaa* ("of temple") resulting in a sentence S_f.

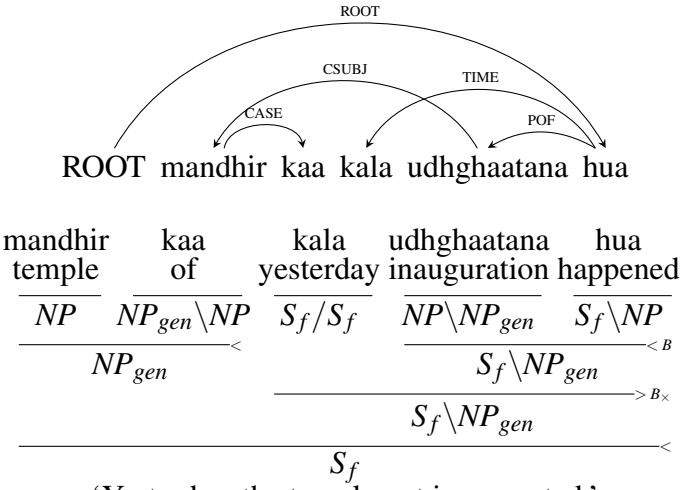


Figure 2.21: Dislocated/Discontinuous genitives (time expression).

Figure 2.22 is similar to Figure 2.21, except that the noun with genitive marker *budhdhiimattaa kii* ("intelligence") is in COBJ dependency relation with the noun of the conjunct verb *taariiph* ("appreciate"). Also the intervening node *jamkara* ("greatly") which is the cause for the crossing arc is an adverb (ADV) unlike the time expression in the previous case.

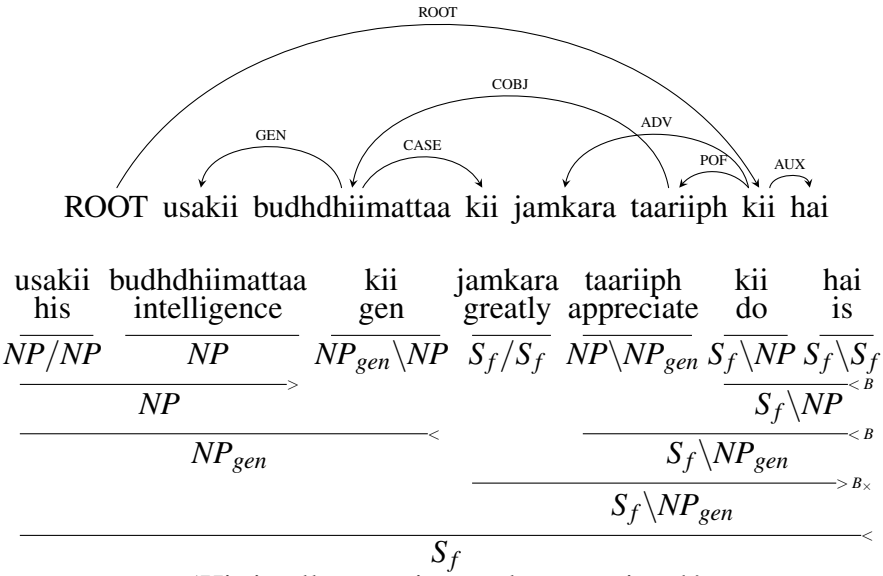


Figure 2.22: Dislocated/Discontinuous genitives (adverb).

2.7.6 Others

Other major dependency labels/constructions which lead to crossing dependency arcs are time/place expressions (TIME/PLACE), noun modifiers (NMOD), SUBJ. These labels corresponds to 9% of crossing arcs.

Similar to adverbs, time/place expressions, due to freer word order nature of Hindi, can occur at any place in the sentence and can be handled using crossed composition in general cases. But, when these occur between nouns in genitive relation or in the conjunct verbs constructions (as in 2.7.5), they lead to crossing arcs, and are handled as discussed in section 2.7.5.

NMOD is the label for noun modifier. NMOD constructions which lead to crossing arcs are similar to those of genitives as in 2.7.5. SUBJ constructions also engender crossing arcs similarly to the OBJ constructions/topicalization in 2.7.3. These constructions are handled similar to the ones described in the previous sections.

2.8 Analysis of the Hindi CCGbank

In this section, we provide a brief analysis of the different CCG categories and combinators in the Hindi CCGbank. Table 2.2 lists the top 12 most frequent CCG categories in both coarse-grained and fine-grained versions of the lexicon. The most com-

CCG Category	Percentage (%)
NP	28.09
NP/NP	16.45
$S_f\backslash S_f$	9.05
NP\NP	6.99
$(S_f/S_f)\backslash NP$	6.66
(NP/NP)\NP	4.53
S_f/S_f	2.56
$(S_f\backslash NP)\backslash NP$	2.21
JJP	2.11
$S_f\backslash NP$	2.05
(NP/NP)/(NP/NP)	1.90
CCP/S_f	1.60

CCG Category	Percentage (%)
NP	17.67
NP/NP	16.44
NP[0]	9.11
$S_f\backslash S_f$	9.05
$(S_f/S_f)\backslash NP$	5.91
(NP/NP)\NP	4.09
S_f/S_f	2.56
JJP	2.12
(NP/NP)/(NP/NP)	1.90
NP[0_ne]\NP	1.84
$S_f\backslash NP[0]$	1.82
NP[0_ko]\NP	1.77

Table 2.2: Distribution of CCG categories in coarse-grained (left) and fine-grained (right) lexicon.

mon categories are the category for nouns (*NP*) and noun modifiers like adjectives and determiners (*NP/NP*). The next most frequent categories are the categories for post-position markers for nouns and auxiliary or tense, aspect and modality (TAM) markers for verbs. $S_f\backslash S_f$ and *NP\NP* are the categories for auxiliary or TAM markers for verbs and post-position markers for nouns respectively. The post-position marker of an adjunct noun phrase gets the category $(S_f/S_f)\backslash NP$. *(NP/NP)\NP* is the category for both genitive marker and conjunction in NP coordination. $(S_f\backslash NP)\backslash NP$ and $S_f\backslash NP$ are the categories for transitive and intransitive verbs respectively. Adjectival phrase gets a category *JJP*. *(NP/NP)/(NP/NP)* is the category for modifier of a noun modifier and CCP/S_f is the category for subordinate conjunction.

Categories in the top 12 list of the fine-grained lexicon but not in the coarse-grained are *NP[0]*, *NP[0_ne]\NP* and *NP[0_ko]\NP*. In this lexicon, the coarse category for nouns gets split into *NP* (the category for a noun with a separate lexical item as a case marker) and *NP[0]* (the category for a noun without any case marker). For example, in noun chunks *raam ne* (“Ram ERG”) and *raam* (“Ram”), the category of *raam* is *NP* in first case and *NP[0]* in the later case. 0 here means that the case marker appeared as a separate lexical item. For example, *raam ne* (“Ram ERG”) will have `NP[0_ne]` as the category whereas *usne* (“he+ERG”) will have `NP[ne]` as the category. This is the

notation followed in the Hindi dependency treebank. The remaining two categories, *NP[0_ne]\NP* and *NP[0_ko]\NP*, are the categories for ergative ('ne') and dative ('ko') case-markers.

Table 2.3 shows the distribution of different CCG combinators in the Hindi CCGbank. Since Hindi is a verb final language, the backward application and composition combinators are more frequent than forward application and composition combinators. Due to freer word order nature and crossing dependency arcs, there are around 0.5% of crossed composition combinators in the Hindi CCGbank. This shows the importance of crossed composition combinators for freer word order languages.

CCG Combinator	**Percentage (%)**
Forward Application (>):	38.61
Backward Application (<):	45.90
Forward Composition (> B):	0.01
Backward Composition (< B):	14.99
Forward Crossed Composition ($> B_X$):	0.04
Backward Crossed Composition ($< B_X$):	0.45

Table 2.3: Distribution of combinators in the Hindi CCGbank.

2.9 Conclusion

We presented an approach for automatically creating a CCGbank from a dependency treebank for Hindi which is a morphologically rich, freer word order and verb final language. We created two types of lexicon: fine-grained which keeps morphological information in noun categories and coarse-grained which doesn't. We have provided a detailed analysis of various long-range dependencies like coordinate and relative constructions, and shown how to handle them in CCG. We have also discussed in detail the different word orders that arise from the free word order nature of Hindi in various constructions, and provided a unified projective analysis for them under CCG. We have also provided a brief analysis of the different CCG categories and combinators in the Hindi CCGbank.

The approach described here has already been successfully applied to Telugu, another Indian language (Kumari and Rao, 2015). In future we would like to extract CCG lexicons and/or CCGbanks for the many other languages for which dependency tree-

banks are available, including the languages of the CoNLL dependency parsing shared tasks (Buchholz and Marsi, 2006; Nivre et al., 2007a) and universal dependency treebanks (McDonald et al., 2013). We would also like to see the impact of generalisation of our lexicon using the free-word order formalism for CCG categories of Baldridge (2002).

In the next chapter, we develop a CCG supertagger for Hindi and show that informative CCG categories, which contain both local subcategorization information and capture long distance dependencies elegantly, improve the performance of dependency parsers.

Chapter 3

Improving Dependency Parsers using CCG Supertags

Subcategorization information is a useful feature in dependency parsing. In this chapter, we explore a method of incorporating this information via Combinatory Categorial Grammar (CCG) categories from a supertagger. We experiment with two popular dependency parsers (Malt and MST) for two languages: English and Hindi. For both languages, CCG categories improve the overall accuracy of both parsers by around 0.3-0.5% in all experiments. For both parsers, we see larger improvements specifically on dependencies at which they are known to be weak: long distance dependencies for Malt, and verbal arguments for MST. Parts of this chapter are based on the content from Ambati et al. (2013) and Ambati et al. (2014).

3.1 Introduction

Dependency parsers can recover much of the predicate-argument structure of a sentence, while being relatively efficient to train and extremely fast at parsing. Dependency parsers have been gaining in popularity in recent times due to the availability of large dependency treebanks for several languages and parsing shared tasks (Buchholz and Marsi, 2006; Nivre et al., 2007a; Bharati et al., 2012).

In this chapter, we show that using CCG categories improve both popular dependency parsers, Malt and MST for two typologically diverse languages, Hindi and English. CCG lexical categories contain subcategorization information regarding the dependencies of predicates, including long-distance dependencies. We show that providing this subcategorization information in the form of CCG categories can help

both Malt and MST on precisely those dependencies for which they are known to have weak rates of recovery. The result is particularly interesting for Malt, the fast greedy parser, as the improvement in Malt comes without significantly compromising its speed, so that it can be practically applied in web scale parsing. Our results apply both to English, a fixed word order and morphologically simple language, and to Hindi, a free word order and morphologically rich language, indicating that CCG categories from a supertagger are an easy and robust way of introducing lexicalized subcategorization information into dependency parsers.

3.2 Related Work

Parsers using different grammar formalisms have different strengths and weaknesses, and prior work has shown that information from one formalism can improve the performance of a parser in another formalism.

Sagae et al. (2007) used the output of a dependency parser to improve a Head-driven Phrase Structure Grammar (HPSG) parser. They achieved a 1.4% improvement in accuracy over a state-of-the-art HPSG parser by using dependencies from a dependency parser for constraining wide-coverage rules in the HPSG parser. Coppola and Steedman (2013) incorporated higher-order dependency features into a cube decoding phrase-structure parser. They experimented with both in-domain and out-of-domain test sets and obtained significant gains in dependency recovery in both cases.

Kim et al. (2012) improved a CCG parser using dependency features. They extracted n-best parses from a CCG parser and provided dependency features from a dependency parser to a re-ranker. They explored four different dependency schemes: CoNLL[1] (Nivre et al., 2007a), Stanford [2] (de Marneffe et al., 2006), LTH [3] (Johansson and Nugues, 2007) and Fanse [4] (Tratz and Hovy, 2011). They used two widely used dependency parsers: Malt and MST for their experiments. Dependency features from MST parser using CoNLL scheme gave better improvements for CCG parsing. In the case of Malt, using Fanse dependency scheme gave the best results. They obtained a final improvement of 0.35% in labelled F-score on the CCGbank test set using features from Malt parser.

Conversely, in this chapter, we show that dependency parsers can be improved by

[1] http://stp.lingfil.uu.se/~nivre/research/Penn2Malt.html
[2] http://nlp.stanford.edu/software/lex-parser.shtml
[3] http://nlp.cs.lth.se/software/treebank_converter/
[4] http://www.isi.edu/publications/licensed-sw/fanseparser/

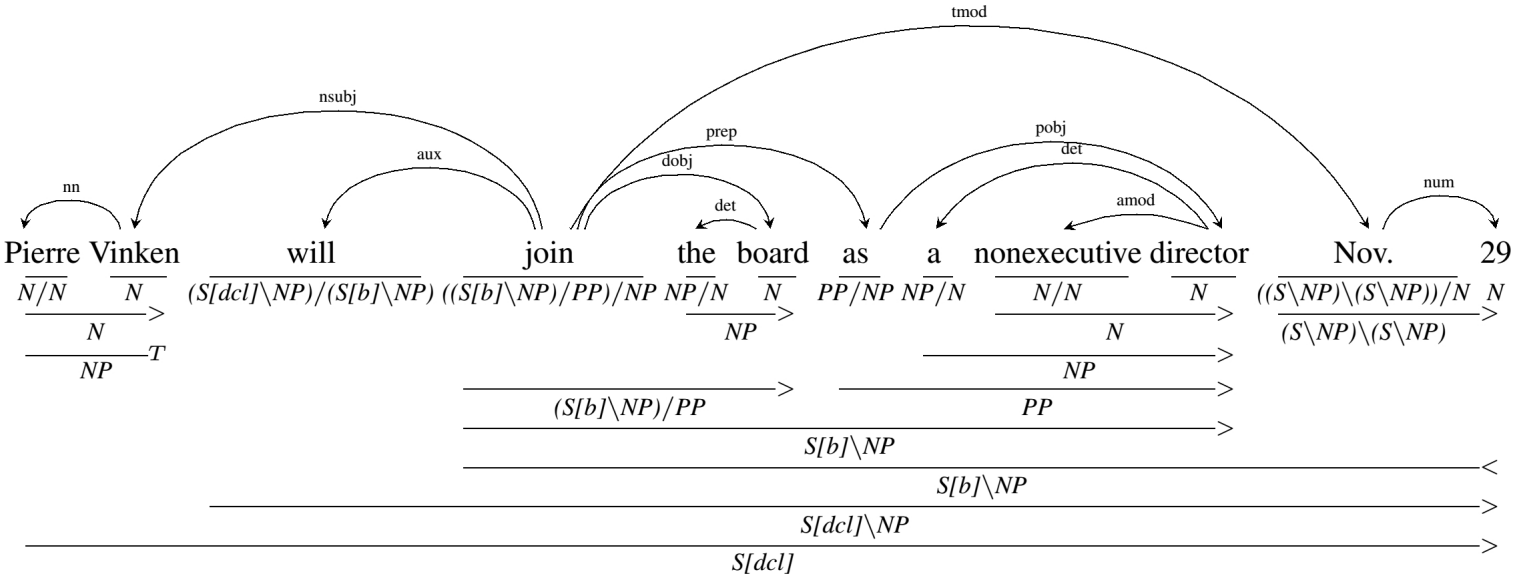


Figure 3.1: An example CCG derivation and the Stanford scheme dependencies.

using CCG categories. There is a little work on using supertags as features to dependency parsing (Foth et al., 2006; Çakıcı, 2009; Ouchi et al., 2014). Foth et al. (2006) improved a constrained based dependency parser for German using supertags. They first designed supertags based on dependency information like dependency labels and directionality. They created ten different versions of supertags with different granularity. Coarsest tagset has 35 tags where as the finest one has 12,947 tags. Using TnT tagger[5], a hidden Markov model based tagger, they developed a supertagger. They used the automatic supertags assigned by this supertagger as a feature in their weighted constrained dependency parser. They obtained significant improvements of around 2.2% in unlabelled attachment score and 2.6% in labelled attachment score with an error reduction of over 24%.

Ouchi et al. (2014) did similar experiments where they improved English dependency parser using supertags. They designed two versions of supertags: coarse-grained and fine-grained, which are based on the dependency tree. Coarse-grained version contains information about the dependency label of the current node, direction of the current node with respect to its parent and also the directions of its children. In the fine-grained version, in addition to the above information, dependency labels for the children are also added. Coarser version has 79 tags whereas the finer version has 312 tags. They developed a Conditional Random Field (CRF) (Lafferty et al., 2001) supertagger. Using the automatic supertags from this supertagger as a feature to the dependency parser, they obtained 1.3% improvement in unlabelled attachment score using coarse-grained supertags as features. The supertag design of Ouchi et al. (2014) is inspired from that of Foth et al. (2006). But Foth et al. (2006) experimented with weighted constraint parser for German, whereas Ouchi et al. (2014) experimented with a data-driven transition-based parser for English.

Supertags used by Foth et al. (2006) and Ouchi et al. (2014) are based on the information from the dependency tree. Çakıcı (2009) experimented with using CCG supertags for MST parser for Turkish. Since the supertagger accuracy is very low for Turkish, they couldn't obtain any improvements in dependency parsing. Unlike Foth et al. (2006) and Ouchi et al. (2014), but similar to Çakıcı (2009), we explored CCG supertags rather than the supertags based on dependencies in our experiments. Also, we experiment with two diverse languages: English and Hindi, and two popular dependency parsers: Malt, a transition-based parser and MST, a graph-based parser. Ouchi et al. (2014) only present unlabelled results with transition-based parser for

[5] http://www.coli.uni-saarland.de/~thorsten/tnt/

English, whereas we present both unlabelled attachment score and labelled attachment score in our experiments. Kumari and Rao (2015) applied our approach to Telugu, another Indian language, whose dependency annotation scheme is based on that of Hindi. They developed a supertagger for Telugu and used supertags as features for Malt and MST parsing models for Telugu. They also obtained similar improvements as our Hindi experiments.

3.3 Grammar Formalisms

In this chapter, we deal with two grammar formalisms: Combinatory Categorial Grammar (CCG) and Dependency Grammar. We provided detailed description of CCG in section 1.2. In this section we present a brief introduction to dependency grammar. For a detailed description about CCG and dependency parsing, readers can refer to Steedman (2000) and Kübler et al. (2009).

Dependency Grammar (DG) describes the syntactic structure of a sentence through dependency graphs. A dependency graph of a sentence represents words and their relationship to syntactic modifiers using directed edges. These edges can be labelled with grammatical relations like Subject, Object etc.

Dependency trees can either be projective or non-projective. Due to English's rigid word order, projective trees are sufficient to analyze most English sentences. But, in languages with free word order, such as Czech, Dutch, German, Hindi etc. non-projective dependencies are more frequent. Rich inflection systems reduce the demands on word order, leading to non-projective dependencies (McDonald, 2006).

CCG categories contain subcategorization information whereas dependency graphs do not have this information explicitly. Unlike CCG, where a derivation is first required to extract word-word dependencies, DG captures dependencies between words directly. CCG derivations are constrained by the CCG rules used for combining categories. But in DG, dependencies can occur between any words within a sentence. CCG captures long distance dependencies elegantly, which can't be done easily in DG (Kim et al., 2012). Figure 3.1 shows a CCG derivation with CCG lexical categories for each word and Stanford scheme dependencies (de Marneffe et al., 2006) for an example English sentence.

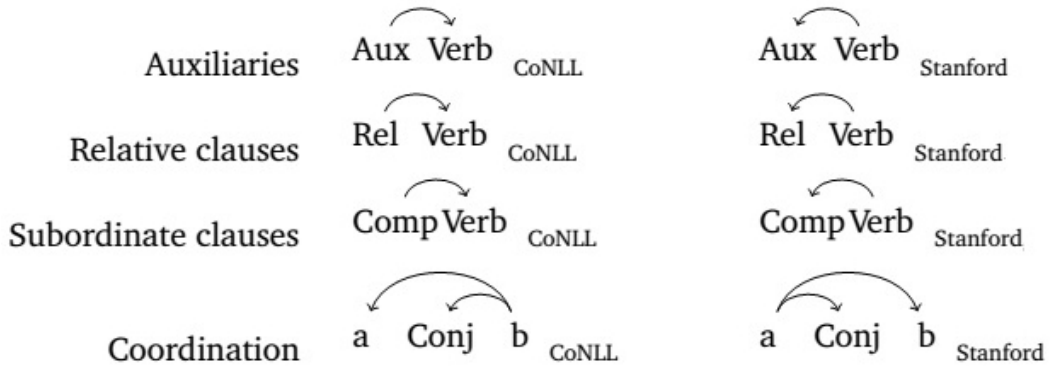


Figure 3.2: Analyses of different constructions in Stanford and CoNLL schemes (adapted from Kim et al. (2012)).

3.4 Data and Tools: English

In this section we describe the dependency and CCG resources available for English.

3.4.1 Treebanks

In English dependency parsing literature, two different dependency schemes, namely, Stanford and CoNLL are widely popular. Stanford scheme, introduced by de Marneffe et al. (2006), is the dependency scheme used in the Stanford parser[6]. CoNLL scheme is the scheme used in the CoNLL dependency parsing shared tasks (Buchholz and Marsi, 2006; Nivre et al., 2007a). Figure 3.2 demonstrates some of the differences between the two dependency schemes. For instance, auxiliaries take the lexical verb as a dependent in CoNLL scheme whereas for Stanford the lexical verb is the head of a verb phrase. Also, CoNLL scheme is the widely explored scheme in English dependency parsing literature, while Stanford scheme has a much richer label set of 48 labels (compared to 11 labels of CoNLL scheme).

We experimented with both these schemes. We used Penn Treebank (Marcus et al., 1993) standard splits, training (sections 02-21), development (section 22) and testing (section 23) for our experiments. We used the Stanford parser's built-in converter with the basic projective option to convert Penn Treebank trees into Stanford scheme dependency trees. For CoNLL scheme, we used Penn2Malt[7], a publicly available tool, to generate CoNLL dependencies from Penn Treebank.

[6]http://nlp.Stanford.edu/software/lex-parser.shtml
[7]http://w3.msi.vxu.se/ nivre/research/Penn2Malt.html

3.4.2 Supertagger

Clark and Curran (2004b) (C&C) developed a Maximum Entropy based CCG supertagger for English using the English CCGbank (Hockenmaier and Steedman, 2007). Using a frequency cutoff of 10, a category set of 409 category types was created. Different features like word, part-of-speech, contextual and complex bi-gram features were explored. The 1-best accuracy of this supertagger on the CCGbank development set is 91.5%. We use this C&C supertagger for our experiments.

3.4.3 Dependency Parsers

We used Malt and MST parsers for our experiments. MaltParser is a freely available implementation of the parsing models described in Nivre et al. (2007b)[8]. It is a classifier based shift reduce parser. With MaltParser, parsing can be performed in linear time for projective dependency trees and quadratic time for arbitrary (possibly non-projective) trees. MaltParser provides options for nine deterministic parsing algorithms: Nivre arc-eager, Nivre arc-standard, Covington projective, Covington non-projective, Stack projective, Stack swap-eager, Stack swap-lazy, Planar and 2-planar. It also provides options for libsvm and liblinear learner algorithms.

MSTParser is a freely available implementation of the parsing models described in McDonald (2006)[9]. It is a graph-based parsing system in which parsing algorithm is equated to finding directed maximum spanning trees from a dense graph of the sentence. MSTParser uses Chu-Liu-Edmonds maximum spanning tree algorithm (Chu and Liu, 1965; Edmonds, 1967) for non-projective parsing and Eisner's algorithm (Eisner, 1996b) for projective parsing. It uses online large margin learning as the learning (McDonald et al., 2005a). It also provides options for 1st order and 2nd order features. 1st order features are the features over the parent and child in the dependency arc. These include different unigram, bigram features of parent node and child node. But, 2nd order features include more global features like grand parent, grand child and sibling features. For example, postag of parent node and child node are 1st order features. Whereas, postag of grand child and grand parent are second order features. Since Malt uses a local model, it is good at short range dependencies. MST is good at long range dependencies since it uses a global model.

There has been a significant amount of work on parsing English using Malt and

[8] http://www.maltparser.org/

[9] http://mstparser.sourceforge.net/

MST parsers in the recent past (Nivre et al., 2007a). We first run these parsers with previous best settings (McDonald et al., 2005b; Foster et al., 2011; Zhang and Nivre, 2012) and treat them as our baseline. In the case of English, Malt uses the arc-standard parsing algorithm for CoNLL scheme and stack-projective algorithm for Stanford scheme. For learning, liblinear learner is used for both the schemes. MST uses 1st-order features, projective parsing algorithm with 5-best MIRA training for both the schemes.

For English, POS-tags are assigned using a perceptron tagger (Collins, 2002), with an accuracy of 97.3% on a standard Penn Treebank test set. Following Zhang and Nivre (2012), we assign automatic POS-tags to the training data using ten-way jackknifing.

3.5 Data and Tools: Hindi

In this section we describe the dependency and CCG resources available for Hindi.

3.5.1 Treebanks

For Hindi, we work with the Hindi Dependency Treebank (HDT ver-0.5) released as part of Coling 2012 Shared Task on parsing (Bharati et al., 2012). The Hindi treebank contains 12,041 training, 1,233 development and 1,828 testing sentences with an average of 22 words per sentence. More details about the treebank can be found in section 2.3.

3.5.2 Supertagger

Following Clark (2002), we used a Maximum Entropy approach to build our supertagger. We explored different features in the context of a 5-word window surrounding the target word. Details of the development of the Hindi supertagger are provided in section 5.4.2. The 1-best accuracy of the supertagger is 82.92% and 84.40% for fine-grained and coarse-grained lexicon respectively. As the number of category types in fine-grained lexicon (376) are much higher than in coarse-grained (202), it is not surprising that the performance of the supertagger is better for coarse-grained as compared to fine-grained.

3.5.3 Dependency Parsers

For Hindi, we did all our experiments using automatic features (pos, chunk and morphological information) extracted using a Hindi shallow parser [10]. Similar to English, we first run Malt and MST with previous best settings (Bharati et al., 2012) and treat them as our baseline. For Hindi, Malt uses the arc-standard parsing algorithm with a liblinear learner. MST uses 2nd-order features, non-projective parsing algorithm with 5-best MIRA training. We compare and analyze results after adding CCG categories as features with this baseline.

3.6 CCG Categories as Features to Malt and MST

Similar to Çakıcı (2009), instead of using CCG supertags for all words, we used supertags which occurred at least K times in the training data, and backed off to coarse POS-tags otherwise. We experimented with different values of K. For English K=1, i.e., when we use CCG categories for all words, gave the best results. K=15 gave the best results for Hindi. As the data for Hindi is small, providing CCG categories to all the words didn't help due to sparsity issues. But for English, due to the relatively larger amount of data, using CCG categories for all the words worked better than using coarse POS-tag based back off. We explored both Stanford and CoNLL schemes in the case of English and fine and coarse-grained CCG categories in the case of Hindi. All feature and parser tuning is done on the development data.

Since both Malt and MST parsers take the data in CoNLL format as input, we provided CCG categories in the `FEATS` column of the CoNLL format. If `S0` is the top node in the stack, `Q0` is the first node in the input queue and `c` is their corresponding CCG category, then the feature templates used are `S0c, Q0c, S0cQ0c`.

3.6.1 Experiments with Gold Categories

We first provided gold CCG categories extracted from CCGbanks as features to the Malt and MST parsers. We used coarse POS tags for the sentences which don't have a CCG derivation. Unlabelled Attachment Scores (UAS) and Labelled Attachment Scores (LAS) on the test set are shown in Table 3.1. As expected, gold CCG categories boosted UAS and LAS by around 4-7% in all the cases. This clearly shows that the rich subcategorization information provided by CCG categories can help dependency

[10] `http://ltrc.iiit.ac.in/analyzer/hindi/`

Language	*Experiment*	*Malt*			*MST*	
		UAS	*LAS*		*UAS*	*LAS*
English	Stanford Baseline	90.32	87.87		90.36	87.18
	Stanford + Gold CCG	**94.83****	**93.06****		**94.83****	**90.96****
	CoNLL Baseline	89.99	88.73		90.94	89.69
	CoNLL + Gold CCG	**94.24****	**93.71****		**95.35****	**93.76****
Hindi	Baseline	88.67	83.04		90.52	80.67
	Fine Gold CCG	**95.27****	**90.22****		**96.60****	**84.95****
	Coarse Gold CCG	95.26**	90.18**		96.32**	84.71**

Table 3.1: Impact of Gold CCG categories on dependency parsing. McNemar's test, ** = $p < 0.01$.

parsers like Malt and MST. All the improvements on test set are statistically significant (McNemar's test, $p < 0.01$).

3.6.2 Experiments with Supertagger output

Having seen improvements with gold CCG categories, we experimented with using automatic CCG categories from a supertagger as a feature to Malt and MST. We performed different feature and parser tuning experiments on the development data and the settings which gave best results are used for test set. Unlabelled Attachment Scores (UAS) and Labelled Attachment Scores (LAS) on the test set are shown in Table 3.2. Numbers in brackets in the table are percentage of errors reduced. Even with automatic categories from a supertagger, we got significant improvements over the baseline, for all the cases. All the improvements on test set are statistically significant (McNemar's test, $p < 0.05$ for Hindi LAS and $p < 0.01$ for the rest). This shows that the rich subcategorization information provided by automatically assigned CCG categories can help Malt and MST.

For English, in the case of Malt, we could achieve 0.2% and 0.3% improvement in UAS and LAS respectively for Stanford Scheme. For CoNLL scheme, these improvements are 0.4% and 0.5% in UAS and LAS respectively. As Stanford scheme has richer dependency label set compared to CoNLL, we could observe better improvements for CoNLL scheme. In the case of MST, we got around 0.5% improvements in all the cases.

In the case of Hindi, with gold CCG categories, the fine-grained lexicon gave

Language	*Experiment*	*Malt*			*MST*	
		UAS	*LAS*		*UAS*	*LAS*
English	Stanford Baseline	90.32	87.87		90.36	87.18
	Stanford + CCG	**90.56** (2.5)**	**88.16** (2.5)**		**90.93** (5.9)**	**87.73** (4.3)**
	CoNLL Baseline	89.99	88.73		90.94	89.69
	CoNLL + CCG	**90.38** (4.0)**	**89.19** (4.1)**		**91.48** (5.9)**	**90.23** (5.3)**
Hindi	Baseline	88.67	83.04		90.52	80.67
	Fine CCG	88.93** (2.2)	83.23* (1.1)		**90.97** (4.8)**	**80.94* (1.4)**
	Coarse CCG	**89.04** (3.3)**	**83.35* (1.9)**		90.88** (3.8)	80.73* (0.4)

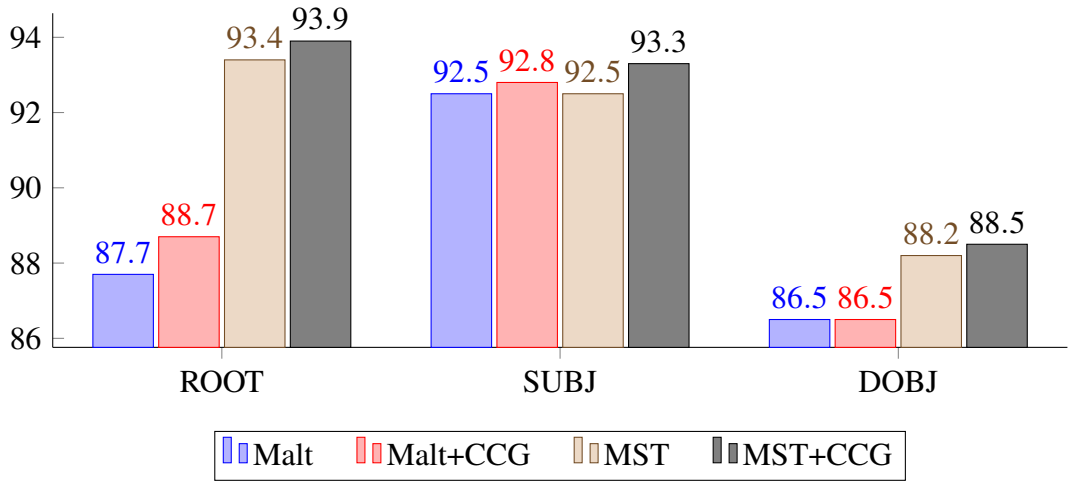
Table 3.2: Impact of CCG categories from a Supertagger on dependency parsing. Numbers in brackets are percentage of errors reduced. McNemar's test, * = $p < 0.05$; ** = $p < 0.01$.

slightly better improvements over coarse-grained as the fine-grained lexicon has richer morphological information. When supertagger output is provided, fine-grained supertags gave better improvements for MST, but for Malt, coarse-grained supertags gave better improvements. The performance of the supertagger on the fine-grained lexicon is slightly lower than that of the coarse-grained lexicon. In the case of Malt, due to local learning, supertagger performance may have led to more error propagation with fine-grained lexicon compared to coarse-grained and hence better performance with coarse-grained supertags. In the case of MST, due to global learning and better handling of error propagation, richer information of fine-grained categories may have surpassed the slight supertagger performance differences. We could achieve final improvements of around 0.3% in both UAS and LAS for Malt. For MST, 0.5% and 0.3% improvement is observed in UAS and LAS respectively.

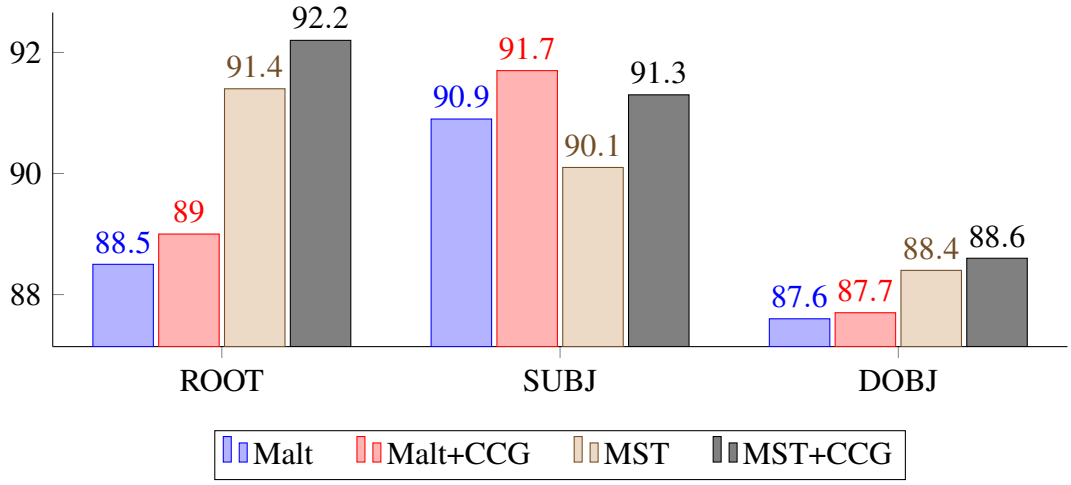
3.6.3 Analysis: English

It is interesting to notice the impact of using automatic CCG categories from a supertagger on long distance dependencies and verbal arguments. It is known that Malt is weak at long-distance relations and MST is weak at verbal arguments (McDonald and Nivre, 2007; Ambati et al., 2010a). Providing CCG categories as features improved handling of long-distance dependencies for Malt and verbal arguments for MST.

Figure 3.3 shows the average F-score of Stanford and CoNLL schemes on the impact of CCG categories for three major dependency labels, namely, `ROOT`, `SUBJ`, `OBJ`, the labels for sentence root, subject, and direct object respectively. For Malt,



(a) CoNLL Dependencies

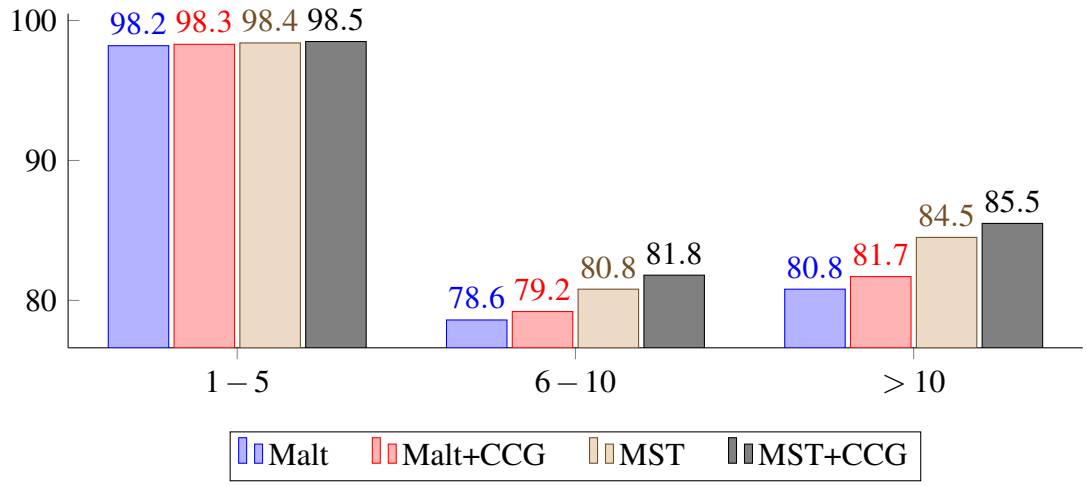


(b) Stanford Dependencies

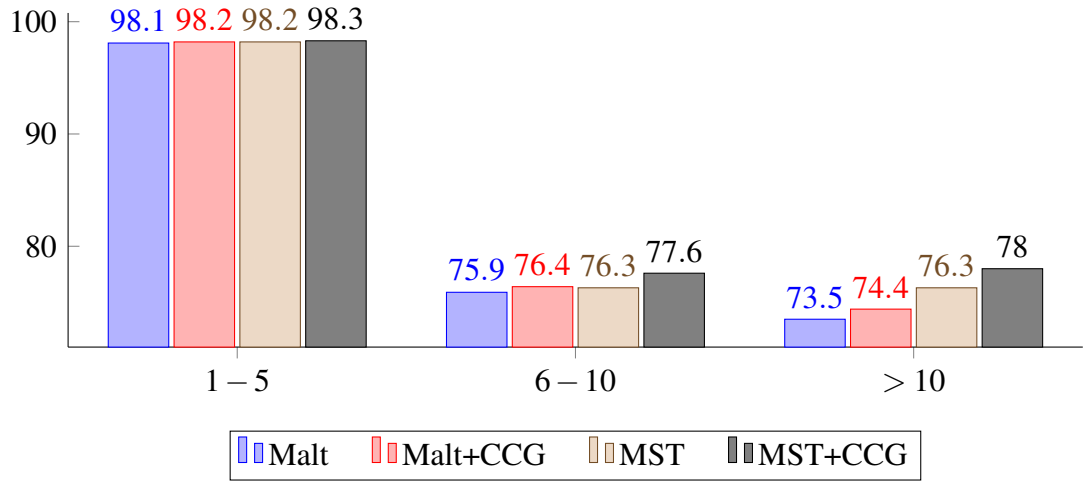
Figure 3.3: Label-wise impact of supertag features for English.

providing CCG categories gave an increment of 0.5%, 0.8% for ROOT, and SUBJ labels respectively over the baseline using Stanford dependencies. For MST, the improvements are 0.8% and 1.2% respectively for ROOT, and SUBJ labels. Similar improvements are observed for CoNLL dependencies as well. There is no significant improvement for direct object label, especially in the case of Malt. This could be because of error propagation, a well known problem with shift-reduce greedy search with local learning parsers. As the CCGbank category for conjunctions in English is 'conj' (as opposed to $(X\backslash X)/X$ which contains subcategorization) which doesn't provide any subcategorization information, there are no significant improvements in the case of co-ordination constructions for English.

We also found that the impact of CCG categories is higher when the span of the



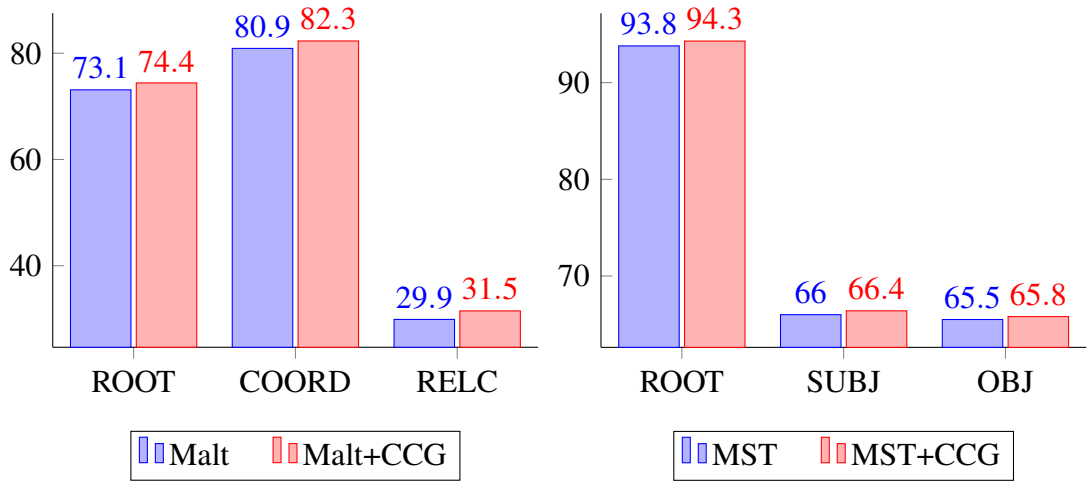
(a) CoNLL Dependencies



(b) Stanford Dependencies

Figure 3.4: Distance-wise impact of supertag features for English.

dependency is longer. Figure 3.4 shows the average F-score of Stanford and CoNLL schemes on the impact of CCG categories for dependencies based on the distance between words. Using CCG categories does not have much impact on short distance dependencies (1−5). For longer range distances, 6−10, and >10, there is significant improvement for both Malt and MST. For Malt, these improvements are 0.5% and 0.9% respectively. In the case of MST, there is an improvement of 1.3% and 1.7% for distances 6−10, and >10 respectively.



(a) Malt Results

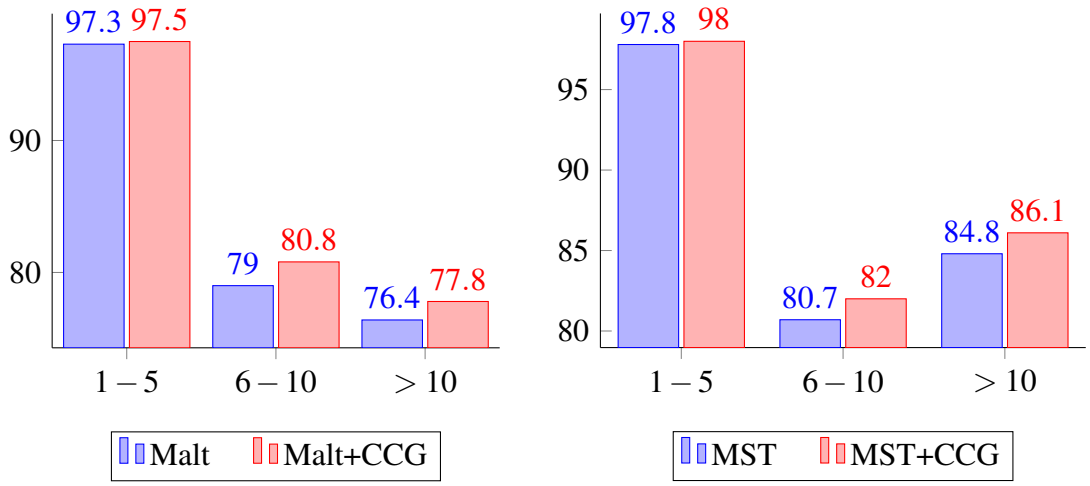
(b) MST Results

Figure 3.5: Label-wise impact of supertag features for Hindi.

3.6.4 Analysis: Hindi

Similar to English, providing CCG categories as features improved the handling of long-distance dependencies for Malt and verbal arguments for MST respectively for Hindi. In the case of Malt, Figure 3.5(a) shows the F-score of the impact of CCG categories on three dependency labels, which take the major share of long distance dependencies, namely sentence root (ROOT), co-ordination (COORD), and relative clause (RELC). For these relations, providing CCG categories gave an increment of 1.3%, 1.4% and 1.6% respectively over the baseline. Unlike English, Hindi CCGbank category of conjunction is $(X\backslash X)/X$ (X depends on the category of the children), which contains subcategorization information. Hence, we observe significant improvements for co-ordination dependencies in Hindi. In the case of MST, Figure 3.5(b) shows the F-score of the impact of CCG categories on sentence root (ROOT), subject (SUBJ) and object (OBJ) verbal arguments. For these relations, providing CCG categories gave an increment of 0.5%, 0.4% and 0.3% respectively over the baseline.

Similar to English, we also observed that the impact of CCG categories is higher when the span of the dependency is longer. Figure 3.6 shows the F-score of the impact of CCG categories on dependencies based on the distance between words. Using CCG categories does not have much impact on short distance dependencies ($1-5$). For longer range distances, $6-10$, and >10, there is an improvement of 1.8% and 1.4% respectively for Malt. In the case of MST, this improvement is 1.3% and 1.3% for distance $6-10$, and >10 respectively.



(a) Malt Results

(b) MST Results

Figure 3.6: Distance-wise impact of supertag features for Hindi.

3.7 Discussion

In this chapter, we showed that CCG lexical categories which contain both local sub-categorization information and capture long distance dependencies elegantly, helped both Malt and MST with both these kinds of dependencies. This result is true for both languages, English (a fixed word order language) for which the two parsers already had high baseline accuracies, and Hindi, which is a free word order and morphologically richer language. For both languages and parsers, CCG categories helped in better recovery of verbal arguments and dependencies when the span of the dependency is longer (6−10, and >10).

Richer CCG categories contain valency information, which has been shown to be very useful feature in dependency parsing both for graph-based and transition-based parsers (Zhang et al., 2013; Zhang and Nivre, 2011). Valency in the form of the number of modifiers of a given head is used by the graph-based sub model of Zhang and Clark (2008) and the models of Martins et al. (2009), and Zhang et al. (2013). Zhang and Nivre (2011) used similar information for their transition-based parser, zpar (Zhang and Clark, 2011b) [11]), which uses beam search and global learning. For English, we also experimented providing CCG categories as features to zpar. But, CCG categories didn't have significant impact in the case of zpar as it already uses similar information.

[11] https://sourceforge.net/projects/zpar/

3.7.1 Impact on Web Scale Parsing

Greedy parsers such as Malt are very fast and are practically useful in large scale applications such as parsing the web. Though valency is a useful feature in dependency parsing, Zhang and Nivre (2012) showed that providing valency information directly didn't help Malt. As Malt can't use valency information, we are providing this information indirectly in the form of CCG categories. We have shown a way to improve Malt without compromising speed and thus enhancing its usefulness for web scale parsing. Table 3.3 shows the speed of Malt, MST and zpar on parsing English testing data in CoNLL scheme. Malt parses around 315 sentences per second, compared to 35 and 11 of zpar and MST respectively. Clearly, Malt is orders of magnitude faster than MST and zpar. After using CCG categories from the supertagger, Malt parsed at the rate of 275 sentences per second, still much higher than other parsers.

Parser	*Ave. Sents / Sec*	*Total Time*
MST	11	3m 36s
zpar	35	1m 11s
Malt	315	0m 7.6s
Malt + CCG	275	0m 9.0s

Table 3.3: Time taken to parse English testing data.

3.8 Conclusion and Future Direction

We have shown that informative CCG categories improve the performance of dependency parsers like Malt and MST. We have shown that both gold CCG categories and automatic categories from a supertagger, added as features to these dependency parsers, help in recovering long distance relations for Malt and verbal arguments for MST in the case of both English and Hindi. This result is particularly interesting in the case of Malt which can't directly use valency information, which CCG categories provide indirectly. This led to an improvement in performance without compromising speed.

Using information from different resources like PropBank (Palmer et al., 2005) and NomBank (Meyers et al., 2004), Honnibal et al. (2010) have created an updated version of CCGbank which includes predicate-argument structures for both verbs and

nouns, baseNP brackets, verb-particle constructions, and nominal modifiers. In future, we would like to explore this CCG resource for English. Though we have worked on English and Hindi, our approach is generic enough to apply to other languages such as Turkish, German etc. for which both dependency and CCG resources are available. Using the algorithms of Ambati et al. (2013) and Cakici (2005), we can extract CCG lexicon and/or CCGbanks for any language with a dependency treebank, including the CoNLL dependency parsing shared task languages, and explore our approach.

Chapter 4

Incremental Parsing for English

We describe a new algorithm for incremental transition-based Combinatory Categorial Grammar parsing in this chapter. We introduce two new actions in the shift-reduce paradigm based on the idea of 'revealing' (Pareschi and Steedman, 1987) the required information during parsing. We present two versions of the incremental parser: a greedy parser which uses a look-ahead and a beam search parser which does not use a look-ahead.

4.1 Introduction

While the majority of CCG parsers use chart-based approaches (Hockenmaier and Steedman, 2002; Clark and Curran, 2007), there has been some work on developing shift-reduce parsers for CCG (Zhang and Clark, 2011a; Xu et al., 2014). Most of these parsers model normal-form CCG derivations (Eisner, 1996a), which are mostly right-branching trees: hence they are not incremental in nature. The dependency models of Clark and Curran (2007) and Xu et al. (2014) model dependencies rather than derivations, but do not guarantee incremental analyses.

Besides being cognitively plausible (Marslen-Wilson, 1973), incremental parsing is more useful than non-incremental parsing for some applications. For example, an incremental analysis is required for integrating syntactic and semantic information into language modeling for statistical machine translation (SMT) and automatic speech recognition (ASR) (Roark, 2001; Wang and Harper, 2003).

In this chapter, we develop a new incremental shift-reduce algorithm for parsing CCG by building a dependency graph in addition to the CCG derivation as a representation. The dependencies in the graph are extracted from the CCG derivation. Since

a node can have multiple parents, we construct a dependency graph rather than a tree. Two new actions are introduced in the shift-reduce paradigm for "revealing" (Pareschi and Steedman, 1987) unbuilt structure during parsing. We build the dependency graph in parallel to the incremental CCG derivation and use this graph for revealing, via these two new actions. On the standard CCGbank test data, our greedy parser achieves improvements of 0.88% in labelled F-score and 2.0% in unlabelled F-score over a greedy non-incremental shift-reduce algorithm. As our algorithm does not model derivations, but rather models transitions, we do not need a treebank of incremental CCG derivations and can train on the dependencies in the existing treebank. Our approach can therefore be adapted to other languages with dependency treebanks, since CCG lexical categories can be easily extracted from dependency treebanks (Cakici, 2005; Ambati et al., 2013). Though we use dependencies for meaning representation and CCG for parsing, our revealing technique can be applied to other meaning representations like lambda expressions and for non-CCG parsing like phrase structure parsing.

The rest of the chapter is arranged as follows. Section 4.2 gives a brief introduction to related work in the areas of CCG parsing and incremental parsing. In section 4.3, we describe our incremental shift-reduce parsing algorithm. Details about the experiments, evaluation metrics and analysis of the results are presented in section 4.4. We conclude with possible future directions in section 4.5.

4.2 Related Work

In this section, we first give a brief introduction to various available CCG parsers. Then we describe approaches towards incremental and greedy parsing.

4.2.1 CCG Parsers

4.2.1.1 Graph-based

There has been a significant amount of work on developing graph-based parsers for CCG. Both generative (Hockenmaier and Steedman, 2002) and discriminative (Clark et al., 2002; Clark and Curran, 2007; Auli and Lopez, 2011; Lewis and Steedman, 2014a) models have been developed. Hockenmaier and Steedman (2002) use generative CKY chart parsing algorithms based on Collins (1997). Clark et al. (2002)'s parser use a conditional model, based on Collins (1996) and a CKY chart parsing algorithm similar to the one described in Steedman (2000). They use the dependency

structures that are derived from the CCG derivations in CCGbank.

Clark and Curran (2007) (C&C) describe a number of log-linear parsing models trained on CCGbank using the CKY chart parsing algorithm. They make considerable use of optimisation techniques and parallelized programming to account for the performance requirement of the estimation task. They present a dependency model and a normal form model and both the models are evaluated by the number of correct dependencies they recover. Curran and Clark (2003) first describe the usefulness of log-linear models for parsing CCG. With log-linear models the parse space can be represented in terms of features, and adding new features is relatively easy. They use Generalised Iterative Scaling (GIS) (Darroch and Ratcliff, 1972) for training. Extending this work, Clark and Curran (2004a) improved both learning and parsing algorithms. They introduced a dependency model which takes dependencies recovered into account in addition to the derivation. Since GIS is inefficient for estimating huge models, a parallel version of the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm (Nocedal and Wright, 1999) was developed.

Fowler and Penn (2010) trained a state-of-the-art Probabilistic Context Free Grammar (PCFG) parser (Petrov and Klein, 2007) on the CCGbank data and obtained encouraging results. Auli and Lopez (2011) experimented with belief propagation and dual decomposition approaches for CCG parsing. They reported the best published results for CCG parsing. Supertag-factored A* CCG parsing was employed by Lewis and Steedman (2014a). They first extract k-best supertags for a sentence. Then they run an A* parser which considers supertag probabilities and CCG combinators to generate a complete spanning analysis. In the absence of a spanning analysis, they increase the beam for the supertagger. In addition to the standard CCGbank test data, they also experimented with two out-of-domain data sets: Wikipedia and BioInfer. Honnibal et al. (2009) annotated 200 Wikipedia sentences for evaluating CCG parsers. Bioinfer is a syntactically annotated corpus of 1,100 sentences from biomedical abstracts (Pyysalo et al., 2007). Though the accuracy of their A* parser is slightly lower than the C&C parser on the CCGbank test set, they obtained significant improvements for out-of-domain data. Because of its simple parsing model, Lewis and Steedman (2014a)'s EasyCCG parser is one of the fastest parsers available for CCG parsing. Lewis et al. (2015) recently introduced a joint A* model for CCG parsing and semantic role labelling. Rather than a traditional pipeline model for semantic role labelling, they present a joint model and showed improvements for both CCG parsing and semantic role labelling.

As these parsers employ a bottom-up chart-parsing strategy and use normal-form CCGbank derivations which are right-branching, they are not incremental in nature. In an SVO (Subject-Verb-Object) language, these parsers first attach the object to the verb and then the subject.

4.2.1.2 Transition-based

Two major works in transition-based CCG parsing with accuracies competitive with the widely used Clark and Curran (2007) parser (C&C) are Zhang and Clark (2011a) and Xu et al. (2014). Zhang and Clark (2011a) used a global linear model trained discriminatively with the averaged perceptron (Collins, 2002) and beam search for their shift-reduce CCG parser. Following Collins and Roark (2004), they apply the early update strategy to perceptron training. Xu et al. (2014) developed a dependency model for shift-reduce CCG parsing using a dynamic oracle technique (Goldberg and Nivre, 2012). Zhang and Clark (2011a) use a beam of size 16 whereas Xu et al. (2014) use a much larger beam of size 128. Unlike the chart parsers, both these parsers can produce fragmentary analyses when a complete spanning analysis is not found. Both these shift-reduce parsers are more incremental than standard chart based parsers. But, as they employ an arc-standard (Yamada and Matsumoto, 2003) shift-reduce strategy on CCGbank, given an SVO language, these parsers are not guaranteed to attach the subject before the object. A detailed description of Zhang and Clark (2011a) style parsing algorithm is described with an example sentence in section 4.3.1.

4.2.2 Greedy Parsers

There has been a significant amount of work on greedy shift-reduce dependency parsing. The Malt parser (Nivre et al., 2007b) is one of the earliest parsers based on this paradigm. The Malt parser implements the transition-based approach to dependency parsing. With this technique, parsing can be performed in linear time for projective dependency trees and quadratic time for arbitrary (possibly non-projective) trees (Nivre and Nilsson, 2005; Nivre, 2008). It provides options for nine parsing algorithms, namely, arc-eager, arc-standard, convington projective, covington non-projective, stack projective, stack eager, stack lazy, planar and 2-planar (Nivre, 2003; Covington, 2001; Nivre and Nilsson, 2005; Nivre, 2009; Gómez-Rodríguez and Nivre, 2010). The parser also provides an option for libsvm[1] (Chang and Lin, 2011) and

[1] http://www.csie.ntu.edu.tw/~cjlin/libsvm

liblinear[2] (Fan et al., 2008) learning models.

Malt is a greedy parser and there have been several extensions made both for better learning and parsing. Goldberg and Nivre (2012) improved learning for greedy parsers by using dynamic oracles rather than a single static transition sequence as the oracle. In all the standard shift-reduce parsers, when two trees combine, only the top node (root) of each tree participates in the action. Sartorio et al. (2013) introduced a technique where in addition to the root node, nodes on the right and left periphery respectively are also available for attachment in the parsing process. They showed significant improvements in performance over the arc-eager and arc-standard algorithms. A non-monotonic parsing strategy was introduced by Honnibal et al. (2013), where an action taken during the parsing process is revised based on future context.

4.2.3 Incremental Parsers

Shift-reduce CCG parsers rely either on CCGbank derivations (Zhang and Clark, 2011a) which are non-incremental, or on dependencies (Xu et al., 2014) which could be incremental in simple cases, but do not guarantee incrementality. Hassan et al. (2009) developed a semi-incremental CCG parser by transforming the English CCGbank into left branching derivation trees. In doing so, they changed the lexical categories for words in order to create fully connected trees. They show that a strictly incremental parser, which conducts only a single pass over the input and uses no look-ahead, performs with very low accuracy. They also show that a semi-incremental parser, which conducts two passes over the input and uses look-ahead, gives a balance between incrementality and accuracy.

There is also some work on incremental parsing using grammar formalisms other than CCG like phrase structure grammar (Collins and Roark, 2004) and tree substitution grammar (Sangati and Keller, 2013). Collins and Roark (2004) developed an incremental parser using a top-down chart parsing algorithm. They used averaged perceptron (Collins, 2002) and beam search for their parser. During training they applied the early update strategy to perceptron training. Sangati and Keller (2013) present an incremental tree substitution grammar parser which is based on the Earley algorithm. They also evaluated their incremental parser for sentence prediction and showed improvements over an n-gram based model.

Now, we discuss five works related to incremental CCG parsing. Hassan et al.

[2] https://www.csie.ntu.edu.tw/~cjlin/liblinear/

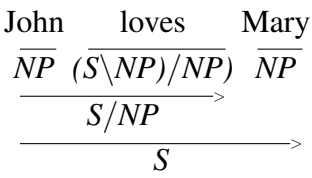


Figure 4.1: Incremental Derivation of a simple sentence

(2009) created an incremental CCGbank and trained a parser on it. Demberg (2012) discussed different constructions which pose problems in converting normal-form derivations to incremental derivations. Pareschi and Steedman (1987) introduced a lazy chart parsing strategy for CCG using unification. Dalrymple et al. (1991) showed a new method of handling ellipses using higher-order unification which can be useful for producing incremental analyses. Kwiatkowski et al. (2010) described a method to automatically map sentences to logical forms using CCGs and higher-order unification.

4.2.3.1 Hassan et al. (2009)

Hassan et al. (2009) developed a semi-incremental CCG parser. They first transformed the English CCGbank into left branching derivation trees. Then they explored incremental left-to-right dependency parsing using the transformed CCGbank. According to Hassan et al. (2009), a strictly incremental parser would conduct only a single pass over the input, use no lookahead and make only local decisions at every word. They showed that such a parser suffers heavy loss of accuracy. They also showed that a semi-incremental (two-pass), linear-time parser that employs fixed and limited look-ahead exhibits an appealing balance between the efficiency advantages of incrementality and the achieved accuracy.

In Hassan et al. (2009), creation of an incremental CCGbank is a crucial step. While creating a left branching tree, they apply the *application* rule for simple categories and *composition* rule for complex categories. For long range dependencies, type-raising followed by forward application (TRFA) is applied. Figure 4.1 gives an incremental derivation of a simple sentence. In this sentence 'John loves Mary', *John* and *loves* are combined by TRFA rule as they can't be combined using *application* or *composition* rules. This results in a category S/NP which is combined with *Mary* using the *application* rule to yield *S*.

This approach works well for simple sentences. But complex constructions like wh-movement, co-ordination etc. pose problems. They introduce new rules and new

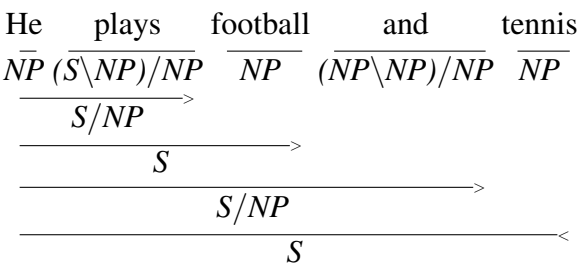


Figure 4.2: Incremental Derivation of a sentence with NP Co-ordination

categories to handle such special cases. Some of the notable changes are:

- All noun categories (N) are converted to noun phrase categories (NP). For example, NP/N is changed to NP/NP.
- Category of conjunction is changed from *conj* to $(X\backslash X)/X$. This implies that the category of a conjunction in NP co-ordination will be $(NP\backslash NP)/NP$ instead of *conj*.
- A new rule called *COORD* is introduced to handle co-ordination constructions (see Figure 4.2). When a conjunction is encountered, they backtrack and extract the suitable category in the context.
- A new rule called *WHMV* is introduced to handle wh-movement.

  Thus, in Hassan et al. (2009)'s work, new rules and/or new categories are introduced as required to create incremental left branching CCG derivation trees. Once we have such a treebank, we can get incremental analysis for a sentence from a left to right parser trained on this treebank.

4.2.3.2 Demberg (2012)

Demberg (2012) describes the complexities involved in creating fully connected left branching trees using CCG. She reports that incremental derivations are not straightforward in the case of CCG. Type-raising is a step towards that, but it cannot generate incremental derivations all the time. She points out that the *Geach Rule* can be useful in creating left branching trees.

$$Geach\ Rule : Y/Z => (Y/G)/(Z/G)$$

But sometimes, we might have to change the category of the word. For example, in an object relative clause construction like "*The woman that every man saw laughed*",

Figure 4.3(a) shows the normal form derivation. Using type-raising we can generate a derivation as in Figure 4.3(b). The most incremental derivation that can be obtained using both type-raising and Geach rule is given in Figure 4.3(c). Demberg (2012) shows that we can achieve a fully incremental derivation only by changing the category of the relative pronoun (4.3(d)). She also mentions that a fully incremental derivation for complement clauses like *Ann thinks the man slept* can't be produced even after using type-raising and Geach rule.

She summarizes that type-raising and Geach rule can help in generating a fully incremental derivation for a few constructions only. But, there still will be some constructions where changing the category of a word would help and some other constructions where we cannot have an incremental derivation with all the available tricks. In this chapter, we show that though CCG is not word by word incremental, it is incremental enough for practical applications.

4.2.3.3 Dalrymple et al. (1991)

Dalrymple et al. (1991) introduce a new method of handling ellipses using higher-order unification. This approach can be used to produce incremental analyses for CCG. An elliptical construction has two clauses (source clause and target clause) which are parallel in structure. The source clause is complete, whereas the target clause has missing material found overtly in the source. For example, consider a verb phrase (VP) ellipsis construction shown below.

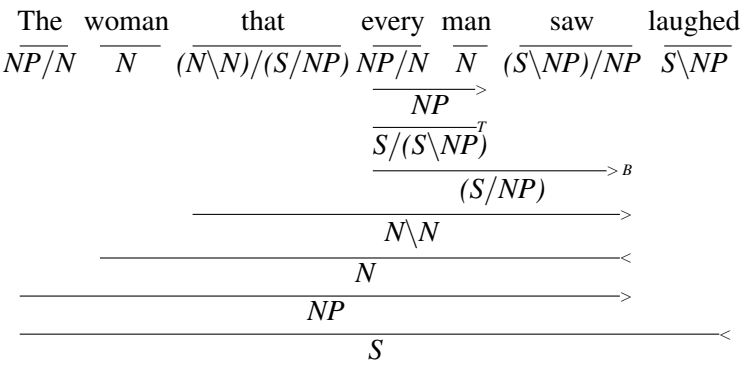
Dan likes golf, *and George does too*.

The meaning of this sentence is that both Dan and George like golf. Here the source clause is *Dan likes golf* and the target clause *George does too*. The source clause parallels the target where 'Dan' and 'George' are parallel elements and the VP of the target sentence is represented by 'does too'. Dalrymple et al. (1991) propose the following procedure to analyze ellipses.

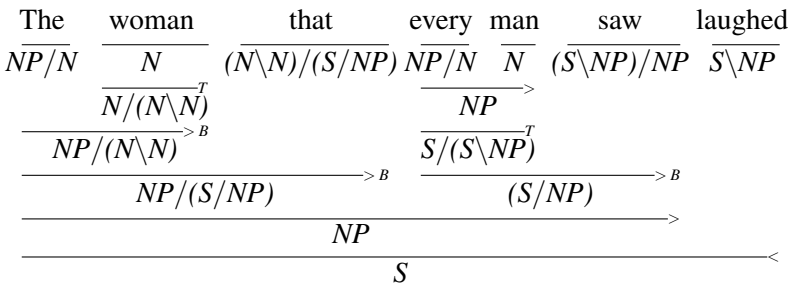
1. Get the analysis of the source clause in the form of parallels

   - P(s1, s2, ... , sn) = s where *si* are parallel elements in the source clause and *s* is the analysis of the source clause.

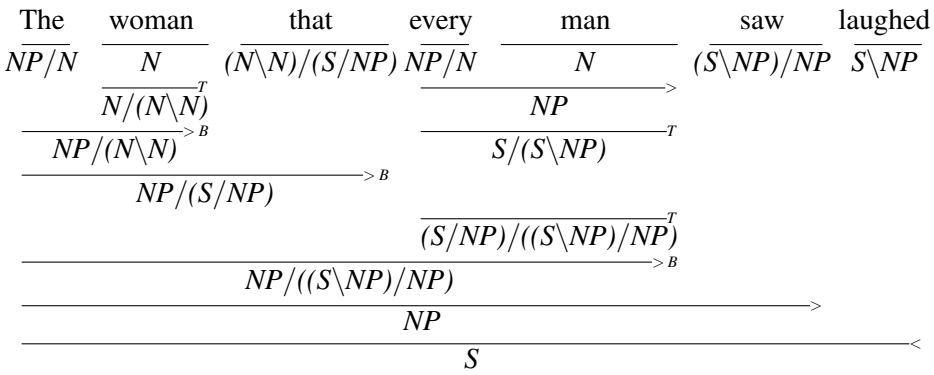
2. Identify the primary occurrences and abstract them out



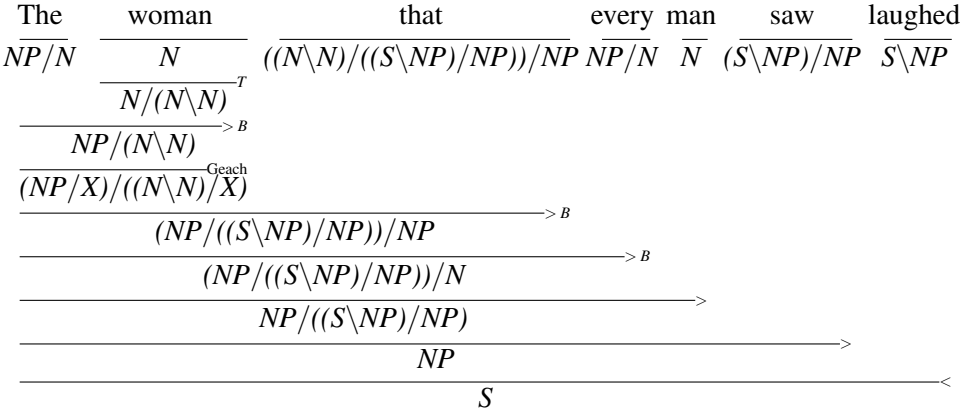
(a) Normal form derivation



(b) Derivation using type-raising



(c) Most incremental derivation using type-raising and Geach rule



(d) Incremental derivation using new category for object relative pronoun

Figure 4.3: CCG derivations for Object Relative Clause construction

- This results in $P-> \lambda x.\lambda y.s(x,y...)$ where x,y etc. are corresponding primary occurrences in the source analysis.

3. Replace the variables with parallels in the target sentence

   - This results in P(t1, t2, ... , tn) = t where *ti* are parallel elements in the target clause and *t* is the analysis of the target clause.

In the above example, analysis of the source clause yields

$$P(dan) = like(dan*, golf)$$

where Dan is the primary occurrence. Abstraction of primary occurrences gives the following

$$P(dan) = \lambda x.like(x, golf)$$

And in the final step, substituting with parallel elements in the target gives

$$P(george) = like(george, golf)$$

We can use a CCG parser or any other semantic analyzer to extract the analysis of the source clause. We can make rules to identify primary occurrences in the source and their parallels in the target based on the type of the sentence or clause (e.g., relative clause, VP co-ordination, ellipses etc.). Higher order unification is used for abstraction. Huet (1975)'s higher-order unification algorithm which provides the solution for generating all possible representations of source (P) can be used. Unification becomes more challenging when there is more than one primary occurrence and/or when the source analysis can be ambiguous. For example, the source analysis for the sentence *"Dan likes his wife, and George does too."* is

$$P(dan) = like(dan*, wife-of(dan))$$

which on abstraction using unification leads to two solutions

$$P = \lambda x.like(x, wife-of(dan))$$

$$P = \lambda x.like(x, wife-of(x))$$

On substitution of parallels, two analyses are possible for the target clause (A) George likes Dan's wife and (B) George likes George's wife.

$$(A) : \lambda x.like(x, wife-of(dan))(george) = like(george, wife-of(dan))$$

$$(B) : \lambda x.like(x, wife-of(x))(george) = like(george, wife-of(george))$$

The task of abstraction becomes more complex when there is more than one primary occurrence. Higher-order unification provides more than one solution which will increase the search space. Context and statistics can be used to prune the solution list to reduce the search space.

4.2.3.4 Pareschi and Steedman (1987)

Pareschi and Steedman (1987) introduced a lazy chart parsing strategy for CCG using unification. Each node is represented using directed acyclic graph feature-structures (FSs) which contain syntax, phonology and semantic information. When two words are connected to produce a new category, the corresponding FSs of these two words are unified. Complex CCG categories are represented through *res*, *arg* and *dir* for result, arguments and their directions.

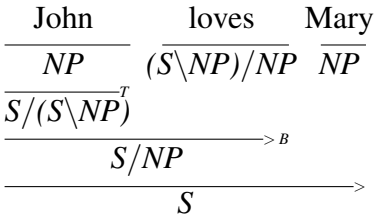
The parsing algorithm of Pareschi and Steedman (1987) is a bottom-up left-to-right algorithm. They build an analysis of the sentence using a shift-reduce strategy keeping only one analysis. When another analysis is required, only at that time, they *reveal* another analysis. The parsing algorithm comprises of four major steps:

1. Scanning: A word is moved to the stack. This is similar to the *SHIFT* action in shift-reduce parsers.

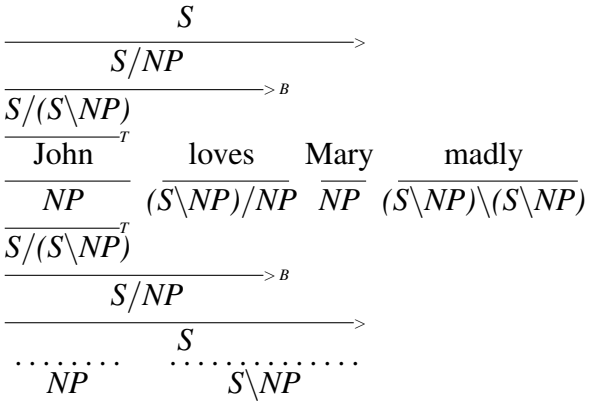
2. Lifting: For every active node in the stack, all unary type-raising rules are applied. This is similar to the *UNARY* action of the Zhang and Clark (2011a)'s shift-reduce CCG parser.

3. Reduce: Two adjacent nodes with categories *X1* and *X2* are reduced to give a category *X0* if there is a CCG rule $X1\ \ X2 => X0$. This is similar to the *BINARY* action of Zhang and Clark (2011a)'s shift-reduce CCG parser.

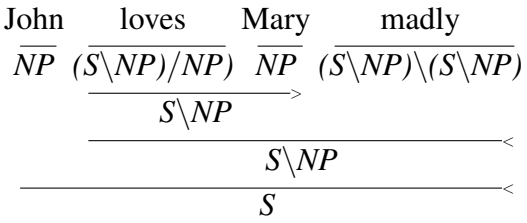
4. Reveal: Let *X1*, $X2\backslash X3$ be the categories of the top two nodes in the stack. In the case of the reveal action *X1* is first split into *X1a* and *X3* given that there is a CCG rule $X1a\ \ X2 => X0$. Now, *X3* and $X2\backslash X3$ are combined to give *X2* which is then combined with *X1a* to give *X0*.



(a) Analysis of a simple sentence



(b) Revealing step



(c) Reduction steps

Figure 4.4: Analysis of a simple sentence using Revealing from Pareschi and Steedman (1987)

Consider a simple sentence "John loves Mary madly". Figure 4.4(a) gives the analysis of this sentence until the 'John loves Mary' part. *John* is type-raised to give a category $S/(S\backslash NP)$ through the *Lifting* operation. This combines with *loves* giving S/NP which combines with *Mary* to give an S. The second derivation in which first *loves* combines with *Mary* and then with *John* is not considered in the initial stage.

When *madly* is shifted to the stack, it needs a VP ($S\backslash NP$) whereas an S is available. The parser chooses the *Reveal* operation. S is split into NP and $S\backslash NP$ with appropriate changes to the feature-structures. Figure 4.4(b) shows this alternative analysis. VP 'loves Mary' now combines with the adverb 'madly' giving $S\backslash NP$. This combines with John (NP) resulting an S as shown in Figure 4.4(c). In this manner, the *Reveal* operation can be used to extract an alternate analysis as and when required. We extend this idea of revealing for our incremental algorithm described in section 4.3.2.

4.2.3.5 Kwiatkowski et al. (2010)

Kwiatkowski et al. (2010) described a method to automatically map sentences to logical forms using CCGs and higher-order unification. Their training data consists of sentences (e.g., 'New York borders Vermont') and their corresponding logical forms (e.g., next_to(ny, vt)). Using this training data they induce the lexicon using the following steps.

- **Initialization**: They start with a multi-word lexicon in the form of $x_i -> S : z_i$ for all training examples (x_i, z_i). For an example sentence 'New York borders Vermont' and its corresponding logical form 'next_to(ny, vt)', the starting item in the lexicon is as below,

$$New\ York\ borders\ Vermont -> S : next\_to(ny, vt)$$

- **Splitting Categories**: A category *X:h* with syntax X and logical form h is split into possible solution pairs (f, g) using higher-order unification. For the above example, where *X:h = S: next_to(ny, vt)* the corresponding f and g can be $f = \lambda x.next\_to(ny, x)$ and $g = ny$.

- **Assigning Syntactic Categories**: Once f and g are extracted, corresponding syntactic categories are assigned. The syntactic category of g is assigned and then all possible categories for f with corresponding directionality are created.

In the above example, NP is assigned to g which leads to two possible categories for f

$$(g = NP : ny \; and \; f = S\backslash NP)$$

$$(f = S/NP \; and \; g = NP : ny)$$

In this work they induce both logical expressions and syntactic categories and extracting logical expressions using higher-order unification is the most important step. Several restrictions were enforced to reduce the number of higher-order solutions. Some of these are incorporated to reduce the computational search space while others are based on the grammar in use.

In this chapter, we develop a new transition-based algorithm for incremental CCG parsing, which is more incremental than Zhang and Clark (2011a) and Xu et al. (2014) and more accurate than Hassan et al. (2009). We also show the impact of beam and look-ahead for incremental parsing. Our algorithm is not strictly incremental as we only produce derivations which are compatible with the Strict Competence Hypothesis (Steedman, 2000) (details in section 4.3.2.3). Unlike Hassan et al. (2009) or Demberg (2012), we do not change any CCG lexical categories. We use the revealing technique of Pareschi and Steedman (1987) for our incremental algorithm. We use a dependency graph for revealing rather than the lambda expressions used by Dalrymple et al. (1991) and Kwiatkowski et al. (2010) since our evaluation is based on the dependency yield of the CCG derivation.

4.3 Algorithms

We first describe the Zhang and Clark (2011a) style shift-reduce algorithm for CCG parsing. Then we explain our incremental algorithm based on the "revealing" technique for shift-reduce CCG parsing.

4.3.1 Non Incremental Algorithm (NonInc)

This is our baseline algorithm and is similar to Zhang and Clark (2011a)'s algorithm (henceforth NonInc). It consists of an input buffer and a stack and has four major parsing actions: `Shift`, `Reduce-Left`, `Reduce-Right` and `Unary`, which are described below in detail with an example sentence.

Since we keep track of dependencies in addition to the CCG derivation, our parser configuration is represented by a triple $< S, I, G >$, where S is the stack, I is the input

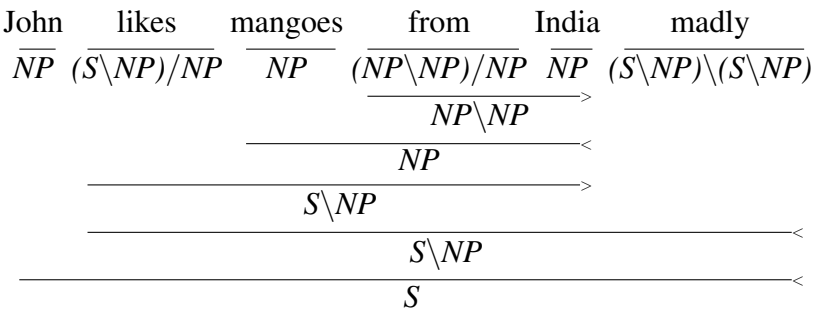


Figure 4.5: Normal form CCG derivation for an example sentence.

buffer and G is the dependency graph. We provide a brief description about each parser action followed by the changes in the parser configuration.

- Shift - X `(S)` : Pushes a word from the input buffer to the stack and assigns a CCG category X. This action performs category disambiguation as well, as X can be any of the categories assigned by a supertagger. If q_1 is the first node in the input, then it is moved to the top of the stack with the category X. Every node in the stack has both the CCG category and the head word information. So, the new node shifted is represented as $X_{(q_1)}$. This action does not add any dependencies to the dependency graph. The parser configuration before (left hand side) and after (right hand side) the shift action is presented below.

$$< S,\ q_1|I,\ G > \quad \rightarrow \quad < S|X_{(q_1)},\ I,\ G >$$

- Reduce Left - X `(RL)` : Pops the top two nodes from the stack, combines them into a new node and pushes it back onto the stack with a category X. This corresponds to binary rules in the CCGbank (e.g., CCG combinators like function application, composition etc., and punctuation rules). In this action the right node is the head and hence the left node is reduced. Let $S1_{(w_1)}$ and $S2_{(w_2)}$ be the top two nodes in the stack, where S_1 and S_2 are the CCG categories and w_1 and w_2 are their corresponding head words. Since the `RL` makes the right node as the head, the head word for category X is w_1. This action also adds a dependency arc from w_1 to w_2.

$$< S|S2_{(w_2)}|S1_{(w_1)},\ I,\ G > \quad \rightarrow \quad < S|X_{(w_1)},\ I,\ G \cup \{w_1 \rightarrow w_2\} >$$

- Reduce Right - X (RR) : This action is similar to the RL (Reduce Left -X) action, except that in this action the right node is reduced since the left node is the head. The parser configuration would be as shown below.

$$< S|S2_{(w_2)}|S1_{(w_1)},\ I,\ G > \quad \rightarrow \quad < S|X_{(w_2)},\ I,\ G \cup \{w_2 \rightarrow w_1\} >$$

- Unary - X (U) : Pops the top node from the stack, converts it into a new node with category X and pushes it back on the stack. The head remains the same in this action. This action corresponds to unary rules in the CCGbank (unary type-changing and type-raising rules). If $S1_{(w_1)}$ is the top node in the stack, then this node becomes $X_{(w_1)}$ after the unary action. Note that this action does not add any additional dependency.

$$< S|S1_{(w_1)},\ I,\ G > \quad \rightarrow \quad < S|X_{(w_1)},\ I,\ G >$$

Figure 4.5 shows a normal-form CCG derivation for an example sentence 'John likes mangoes from India madly'. Figure 4.6 shows the sequence of steps using the NonInc algorithm for parsing the sentence. For simplicity, unary productions leading to *NP* are not described. From step 1 through step 5, the first five words in the sentence (John, likes, mangoes, from, India) are shifted with corresponding categories using shift actions (S). In step 6, *(NP\NP)/NP:from* and *NP:India* are combined using the Reduce-Right (RR) action to form *NP\NP:from* which is combined with *NP:mangoes* in step 7 to form *NP:mangoes*. Step 8 combines *(S\NP)/NP:likes* with *NP:mangoes* to form *S\NP:likes* using the RR action. Then the next word 'madly' is shifted in step 9, which is then combined with *S\NP:likes* in step 10. In step 11, *NP:John* and *S\NP:likes* are combined using the Reduce-Left (RL) action leading to *S:likes*. The parsing process terminates at this step as there are no more tokens in the input buffer and there is only a single node left on the stack.

We use indexed CCG categories (Clark et al., 2002) and obtain the CCG dependencies after every action to build the dependency graph in parallel to the CCG derivation. This is similar to Xu et al. (2014) but differs from Zhang and Clark (2011a), who extract the dependencies at the end after obtaining a derivation for the entire sentence. Figure 4.6 also shows the dependency graph generated and the arc labels give the step ID after which the dependency is generated.

Figure 4.6: NonInc - Sequence of actions with parser configuration and the corresponding dependency graph.

4.3.2 Revealing based Incremental Algorithm (RevInc)

The NonInc algorithm described above is not incremental because it relies purely on the mostly right-branching CCG derivation. In our example sentence, the verb (likes) combines with the subject (John) only at the end (step ID = 11) after all the remaining words in the sentence are processed, making the parse non-incremental. In this section we describe a new incremental algorithm based on a 'revealing' technique (Pareschi and Steedman, 1987) which tries to build the most incremental derivation.

4.3.2.1 Revealing

Pareschi and Steedman (1987)'s original version of revealing was defined in terms of (implicitly higher-order) unification. It was based on the following observation. If we think of categories as terms in a logic programming language, then while we usually think of CCG combinatory rules like the following as applying with the two categories on the left X/Y and Y as inputs, say instantiated as S/NP and NP, to define the category X on the right as S, in fact instantiating *any* two of those categories defines the third.

$$X/Y \quad Y \implies X$$

For example, if we define X and X/Y as S and S/NP, we clearly define Y as NP. They proposed to use unification-based revealing to recover unbuilt constituents from the result of overly-greedy incremental parsing. A related second-order matching-based mechanism was used by Kwiatkowski et al. (2010) to decompose logical forms for semantic parser induction.

The present incremental parser uses a related revealing technique confined to the right periphery and defined over dependency graphs as meaning representation rather than λ-terms. Using CCG combinators and rules like type-raising followed by forward composition, we combine nodes in the stack if there is a dependency between them. However, this can create problems for the newly shifted node as its dependent might already have been reduced. For instance, if the object 'mangoes' in our running example is reduced after it is shifted to the stack, then it will not be available for the preposition phrase (PP) 'from India' (of course, this goes for more complex NPs as well). We have to extract 'mangoes', which is hidden in the derivation, so as to make the correct attachment to the PP. This is where revealing comes into play. 'Mangoes' is "revealed" so that it is available to attach to the PP following it, although it has already been reduced. To handle this, in addition to the four actions of the NonInc

algorithm, we introduce two new actions: Left Reveal (LRev) and Right Reveal (RRev). For this, after every action, in addition to updating the stack we also keep track of the dependencies resolved and update the dependency graph accordingly[3]. In other words, we build the dependency graph for the sentence in parallel to the CCG derivation. As these dependencies are extracted from the CCG derivation, a node can have multiple parents and hence we construct a dependency graph rather than a tree.

- Left Reveal (LRev) : Pop the top two nodes in the stack (left, right). Identify the left node's child with a subject dependency. Abstract over this child node and split the category of left node into two categories. Combine the nodes using CCG combinators accordingly. VP modifiers like VP coordination require this action. Let $S1_{(w_1)}$ and $S2_{(w_2)}$ be the top two nodes in the stack, where S_1 and S_2 are the CCG categories and w_1 and w_2 are their corresponding head words. After the end of the LRev action $S2_{(w_2)}$ will be the top of the stack with a dependency from w_2 to w_1 added to the dependency graph.

$$< S|S2_{(w_2)}|S1_{(w_1)},\ I,\ G > \quad \rightarrow \quad < S|S2_{(w_2)},\ I,\ G \cup \{w_2 \rightarrow w_1\} >$$

- Right Reveal (RRev) : Pop the top two nodes in the stack (left, right). Check the right periphery of the left node in the dependency graph, extract all the nodes with compatible CCG categories and identify all the possible nodes that the right node can combine with. Right periphery of a node is the list of all the right-most nodes in the graph like the right-most child, grand-child, grand-grand-child etc. Select the head node and abstract over this node (e.g. object), split the category into two categories accordingly and combine the nodes using CCG combinators. Constructions like NP coordination, and PP attachment require this action. If $S1_{(w_1)}$ and $S2_{(w_2)}$ are the top two nodes in the stack, then at the end of the RRev action $S2_{(w_2)}$ will be the top of the stack. And the dependency head of w_1 will be a node in the right periphery of S_2, say w_p.

$$< S|S2_{(w_2)}|S1_{(w_1)},\ I,\ G > \quad \rightarrow \quad < S|S2_{(w_2)},\ I,\ G \cup \{w_p \rightarrow w_1\} >$$

[3] Xu et al. (2014) also obtain CCG dependencies after every action. But they do not have a dependency graph which is updated based on the CCG derivation and used in the CCG parsing (in our case for LRev and RRev actions).

(1)	S	[	NP_{John}		
(2)	S	[	NP_{John}	$(S\backslash NP)/NP_{likes}$	
(3)	RL	[	S/NP_{likes}		
(4)	S	[	S/NP_{likes}	$NP_{mangoes}$	
(5)	RR	[	S_{likes}		
(6)	S	[	S_{likes}	$(NP\backslash NP)/NP_{from}$	
(7)	S	[	S_{likes}	$(NP\backslash NP)/NP_{from}$	NP_{India}
(8)	RR	[	S_{likes}	$NP\backslash NP_{from}$	
(9)	RRev	[	S_{likes}		
(10)	S	[	S_{likes}	$(S\backslash NP)\backslash(S\backslash NP)_{madly}$	
(11)	LRev	[	S_{likes}		

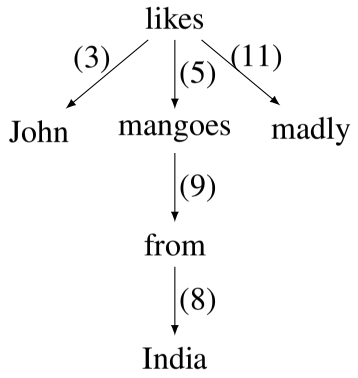


Figure 4.7: RevInc - Sequence of actions with parser configuration and the corresponding dependency graph.

4.3.2.2 Worked Example

Figure 4.7 shows the sequence of steps for the example sentence described above. In steps 1 and 2, the first two words in the sentence: 'John' and 'likes', are shifted from the input buffer to the stack. In addition to standard CCG combinators of application and composition, we also use type-raising followed by forward composition[4]. In step 3, the category of the left node 'John', *NP*, is type-raised to $S/(S\backslash NP)$ which is then combined with the category of right node 'likes', $(S\backslash NP)/NP$, using forward composition operator to yield the category S/NP. This step also updates the dependency graph with an edge between 'John' and 'likes', where 'likes' is the parent and 'John' is the child. The next word 'mangoes' is shifted in step 4 and combined with *S/NP:likes* in step 5 using RR action yielding *S:likes*. After this step, the dependency graph will have 'likes' as the root, with 'John' and 'mangoes' as its children. In this way, as our algorithm tries to be more incremental, both subject and object arguments are resolved as soon as the corresponding tokens are shifted to the stack.

In steps 6 and 7, the next two words 'from' and 'India' are shifted to the stack. Step 8 combines $(NP\backslash NP)/NP$*:from* and *NP:India* using RR action to form $NP\backslash NP$*:from*. Now, we apply the `RRev` action in step 9 to correctly attach 'from' to 'mangoes'. In `RRev` we first check the right periphery and identify a possible node to be attached, 'mangoes', which is the object argument of the verb 'likes'. We abstract over this object and split the category in the following manner: If X is the category of the left node and $Y\backslash Y$ is the category of the right node, then X is split into X/Y and Y with corresponding heads. The head of the left node will be the head of X/Y, and the dependency graph helps in identifying the correct head for Y. Now, Y and $Y\backslash Y$ can be combined using the backward application rule to form Y, which can be combined with X/Y to form X back. In our example sentence, *S:likes* is split into *S/NP:likes* and *NP:mangoes*. *NP:mangoes* is combined with $NP\backslash NP$*:from* to form *NP:mangoes*, which in return combines with *S/NP:likes* and forms back *S:likes*. Figure 4(a) sketches this process. This action also updates the dependency graph with a dependency between 'mangoes' and 'from'.

The next word 'madly' is shifted in step 10, after which the stack has two nodes *S:likes* and $(S\backslash NP)\backslash(S\backslash NP)$*:madly*. We apply the `LRev` action to combine these two nodes. We abstract over the subject of the left node, 'likes', and split the category.

[4]Type-raising followed by forward composition is treated as a single step. Without this, after type-raising, the parser has to check all possible actions before applying forward composition, making it slower.

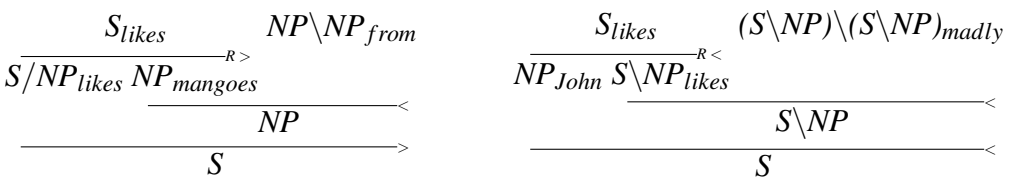


Figure 4.8: RRev and LRev actions.

Here, *S:likes* is split into *NP:John* and *S\NP:likes*. *S\NP:likes* is combined with *(S\NP)\(S\NP):madly* to form *S\NP:likes*, which in return combines with *NP:John* and forms back *S:likes*. The dependency graph is updated with a dependency between 'likes' and 'madly'. Note that the final output is a standard CCG tree. Figure 4(b) shows this LRev action.

4.3.2.3 Analysis

Our incremental algorithm uses a combination of the CCG derivation and a dependency graph that helps to 'reveal' unbuilt structure in the CCG derivation by identifying heads of the revealed categories. For example in Figure 4.8(a), in the `RRev` action, *S:likes* is split into *S/NP:likes* and *NP:mangoes*. The splitting of categories is deterministic but the right periphery of the dependency graph helps in identifying the head, which is 'mangoes'. The theoretical idea of 'revealing' is from Pareschi and Steedman (1987), but they used only a toy grammar without a model or empirical results. Checking the right periphery is similar to Sartorio et al. (2013) and abstracting over the left or right argument is similar to Dalrymple et al. (1991). Currently, we abstract only over arguments. Adding a new action to abstract over the verb as well will make our algorithm handle ellipses in the sentences like 'John likes mangoes and Mary too' similar to Dalrymple et al. (1991) but we leave that for future work.

Currently we use a dependency graph to reveal the unbuilt structure in the CCG derivation. We can also use other representations like lambda calculus instead of dependency graphs. Similar to Kwiatkowski et al. (2010) we can represent the lambda expression in a graph format and use the left and right periphery to reveal the required information similar to dependency graphs. Also our revealing technique is generic enough to be applied to non-CCG parsing like phrase structure parsing. Similar to CCG derivation, we can extract dependencies from the phrase structure tree using head dependency rules (Collins, 1999; de Marneffe et al., 2006).

In practice, we look at the bottom five nodes in the right periphery of the left node in the stack and extract the nodes with a compatible CCG category. If `n` is the number of words in the sentences, then the Complexity of NonInc is $O(n)$. Since we check five extra nodes in the RevInc algorithm, worst-case complexity is $O(5n)$ which is still linear in the sentence length. Also, since both the revealing actions lead to binary derivations, they won't result in cycles.

Our system is monotonic in the sense that the set of dependency relationships grows monotonically during the parsing process. Our algorithm gives derivations almost as incremental as Hassan et al. (2009) but without changing the lexical categories and without backtracking. The only change we made to the CCGbank is making the main verb the head of the auxiliary rather than the reverse as in CCGbank derivations. In the right derivational trees of CCGbank, the main verb is the head for its right side arguments and the auxiliary verb is the head for the left side arguments in the derivation. Not changing the head rule would make our algorithm use the costly reveal actions significantly more, which we avoid by changing the head direction. 3% of the total dependencies are affected by this modification.

Though our algorithm can be completely incremental, we currently compromise incrementality in the following cases:

(a) When there is no dependency between the nodes in the stack.

(b) In the presence of unary type-changing and non-standard binary rules.

(c) In the case of adjuncts like VP modifiers and coordinate constructions like VP, sentential coordination.

We find empirically that extending incrementality to cover these cases actually reduces parsing performance significantly. It also violates the Strict Competence Hypothesis (SCH) (Steedman, 2000), which argues on evolutionary and developmental grounds that the parser can only build constituents that are typable by the competence grammar. We explored the adjunct case of attaching only the preposition first rather than creating a complete prepositional phrase and then attaching it to correct parent. In our example sentence, this would be the case of attaching the preposition 'from' to its parent using `RRev` and then combining the *NP* 'India' accordingly as opposed to creating the preposition phrase 'from India' first and then using `RRev` action to attach it to the correct parent. Though the former is more incremental, it is inconsistent with the SCH. The latter analysis is consistent with strict competence and also gave better

parsing performance while compromising incrementality only slightly. The empirical impact of these differing degrees of incrementality on extrinsic evaluation of our algorithm in terms of language modeling for SMT or ASR is left for future work.

Using our incremental algorithm, we converted the CCGbank derivations into a sequence of shift-reduce actions. We could convert around 98% of the derivations, which is the coverage of our algorithm, recovering around 99% of dependencies. Problematic cases are mainly the ones which involve non-standard binary rules, and punctuations with lexical CCG categories other than 'conj', used as a conjunction, or ',' which is treated as a punctuation mark.

4.4 Experiments and Results

We re-implemented Zhang and Clark (2011a)'s model for our experiments. We used their global linear model trained with the averaged perceptron (Collins, 2002). We applied the early-update strategy of Collins and Roark (2004) while training. In this strategy, when we do not use a beam, decoding is stopped when the predicted action is different from the gold action and weights are updated accordingly.

We use the feature set of Zhang and Clark (2011a) (Z&C) for the NonInc algorithm. This feature set is comprised of features over a) the top four nodes in the stack (S0, S1, S2, S3), b) the next four nodes in the input (Q0, Q1, Q2, Q3) and c) the left and right children of the top two nodes in the stack (S0L, S0R, S1L, S1R) and head and unary head for the top two nodes in the stack (S0H, S0U, S1H, S1U). All the features are based on words (w) and POS-tags (p) and CCG categories (c) for these nodes. For our own model, RevInc, in addition to these features used for NonInc, we also provide features based on the right periphery of the top node in the stack. For nodes in the right periphery, we provide uni-gram and bi-gram features based on the node's CCG category. For example, if S0 is the node on the top of the stack, B1 is the bottom-most node in the right periphery, and c represents the node's CCG category, then B1c, and B1cS0c are the uni-gram and bi-gram features respectively. We extract features based on bottom five nodes (B1, B2, B3, B4, B5) in the right periphery of the top node (S0) in the stack. A complete list of the additional features used for RevInc is presented in Table 4.1. All features except the last block (Type: Graph) are extracted for the NonInc algorithm. In total we used 64 features for NonInc and 72 features for RevInc.

Z&C use a beam of size 16 for their experiments. We first experiment in a greedy setting using a look-ahead of three elements and without the use of a beam. Then we

Type	*Features*
Stack (Basic)	S0wp, S0c, S0pc, S0wc, S1wp, S1c, S1pc, S1wc, S2pc, S2wc, S3pc, S3wc,
Input (Basic)	Q0wp, Q1wp, Q2wp, Q3wp,
Stack (Children)	S0Lpc, S0Lwc, S0Rpc, S0Rwc, S0Upc, S0Uwc, S1lpc, S1lwc, S1Rpc, S1Rwc, S1Upc, S1Uwc,
Bigram (Basic)	S0wcS1wc, S0cS1w, S0wS1c, S0cS1c, S0wcQ0wp, S0cQ0wp, S0wcQ0p, S0cQ0p, S1wcQ0wp, S1cQ0wp, S1wcQ0p, S1cQ0p,
Trigram (Basic)	S0wcS1cQ0p, S0cS1wcQ0p, S0cS1cQ0wp, S0cS1cQ0p, S0pS1pQ0p, S0wcQ0pQ1p, S0cQ0wpQ1p, S0cQ0pQ1wp, S0cQ0pQ1p, S0pQ0pQ1p, S0wcS1cS2c, S0cS1wcS2c, S0cS1cS2wc, S0cS1cS2c, S0pS1pS2p,
Trigram	S0cS0HcS0Lc, S0cS0HcS0Rc, S1cS1HcS1Rc, S0cS0RcQ0p, S0cS0RcQ0w, S0cS0LcS1c, S0cS0LcS1w, S0cS1cS1Rc, S0wS1cS1Rc
Graph	B1c, B1S0c B2c, B2S0c B3c, B3S0c B4S0c, B5S0c

Table 4.1: Feature templates for RevInc.

explore the impact of beam and look-ahead for our incremental parser. When we do not use a look-ahead all the features involving Q1, Q2 and Q3 are excluded. So, for our RevInc, we extract 72 features when we use a look-ahead and extract 64 features when we do not use a look-ahead. Z&C and Xu et al. (2014), use C&C's `generate` script and unification mechanism respectively to extract dependencies for evaluation. C&C's grammar does not cover all the lexical categories and binary rules in the CCGbank. To avoid this, we adapted Hockenmaier's scripts used for extracting dependencies from the CCGbank derivations.

4.4.1 Data and Settings

We use the standard CCGbank training (sections $02 - 21$), development (section 00) and testing (section 23) splits for our experiments. All sentences in the training set are used to train NonInc. But for RevInc, we used 98% of the training set (the coverage of our algorithm). We use automatic POS-tags and lexical CCG categories assigned using the C&C POS tagger and supertagger respectively for development and test data. For training data, these tags are assigned using ten-way jackknifing (Zhang and Clark, 2011a). Also, for lexical CCG categories, we use a multitagger which assigns k-best supertags to a word rather than 1-best supertagging (Clark and Curran, 2004b). The number of supertags assigned to a word depends on a β parameter. Unlike Z&C, the default value of β of 0.01 gave us better results rather than decreasing the value to 0.0001.

Following Z&C and Xu et al. (2014), during training, we also provide the gold CCG lexical category to the list of CCG lexical categories for a word if it is not assigned by the supertagger.

4.4.2 Connectedness and Waiting Time

Before evaluating the performance of our algorithm, we introduce two measures of incrementality: connectedness and waiting time. In a shift-reduce parser, a derivation is fully connected when all the nodes in the stack are connected leading to only one node in the stack before a new node is shifted. We measure the average number of nodes in the stack before shifting a new token from input buffer to the stack, which we call connectedness. For a fully connected incremental parser like Hassan et al. (2009), connectedness would be one. As our RevInc algorithm is not fully connected, this number will be greater than one. For example, in a noun phrase 'the big book', when 'the' and 'big' are in the stack, as there is no dependency between these two words, our algorithm does not combine these two nodes resulting in having two nodes in the stack[5]. The second column in Table 4.2 gives this number for both NonInc and RevInc algorithms. Though our algorithm is not fully connected, connectedness of our algorithm is significantly lower than the NonInc algorithm as our algorithm is more incremental.

We define waiting time as the additional number of nodes that need to be shifted to the stack before a dependency between any two nodes in the stack is resolved. In our example sentence, there is a dependency between 'John' and 'likes'. For

[5]This is a case where the dependencies are not true to the CCG grammar, and make our algorithm less incremental than SCH would allow.

Algorithm	*Connectedness*	*Waiting Time*
NonInc	4.62	2.98
RevInc	**2.15**	**0.69**

Table 4.2: Connectedness and waiting time.

NonInc, this dependency is resolved only after all the four remaining words in the sentence are shifted. In other words, it has to wait for four more words before this dependency is resolved and hence the waiting time is four. On the other hand, in our RevInc algorithm, this dependency is resolved immediately, without waiting for more words to be shifted, and hence the waiting time is zero. The third column in Table 4.2 gives the waiting time for both the algorithms. Waiting time would be zero for a fully connected derivation. Since we compromised incrementality in cases like coordination, waiting time for our RevInc algorithm is not zero but it is significantly lower than the NonInc algorithm and hence more incremental. This property is likely to be crucial for future applications in ASR and SMT language modeling.

4.4.3 Greedy

We trained the perceptron for both the NonInc and RevInc algorithms using the CCGbank training data for 30 iterations, and the models which gave best results on development data are directly used for test data. Table 4.3 gives the unlabelled precision (UP), recall (UR), F-score (UF) and labelled precision (LP), recall (LR), F-score (LF) results of both the NonInc and RevInc approaches on the development data. The last column in the table gives the category accuracy. We used the modified CCGbank for all experiments, including NonInc, for consistent comparisons. For NonInc, the modification decreased unlabelled F-score by 0.45%, without a major difference in labelled F-score.

The top block (first two rows) in Table 4.3 presents the results in the greedy settings. Our incremental algorithm gives an accuracy of 88.69% and 80.75% in unlabelled and labelled F-scores which is an improvement of 1.39% and 0.47% over the NonInc algorithm respectively. For both unlabelled and labelled scores, the precision of RevInc is slightly lower than NonInc but the recall of RevInc is much higher than NonInc resulting in a better F-score for RevInc. As NonInc is not incremental and as it uses more context to the right while making a decision, it makes more precise actions. But, on the other hand, if a node is reduced, it is not available

for future actions. This is not a problem for our RevInc algorithm which is the reason for higher recall. For example, in the example sentence, 'John likes mangoes from India madly', if the object 'mangoes' is reduced after it got shifted to the stack, then in the case of NonInc, the prepositional phrase 'from India' can never be attached to 'mangoes'. But, RevInc makes the correct attachment using the `RRev` action. Category accuracy of NonInc is better than RevInc, since NonInc can use more context before taking a complex action and is less prone to error propagation compared to RevInc.

4.4.4 Beam

To compare these results in the perspective of Z&C's parser we also trained our NonInc and RevInc parsers with a beam size of 16 similar to Z&C. Increasing the beam size increases the accuracy but significantly reduces the parsing speed. Z&C showed that a small beam is sufficient to capture most of the ambiguity in the analyses with reasonable tradeoff between speed and accuracy. The second block (3-4 rows) in Table 4.3 shows these results and the last row presents the results from their paper. Results with our implementation of Z&C are 0.65% lower than the published results, possibly due to the modification made in the head rule, and other minor differences like the supertagger beta value. Unlabelled and labelled F-scores of our RevInc parser are lower than NonInc when we use a beam. Since NonInc is not incremental, it makes use of better context. Also the advantage of the RevInc algorithm in the greedy settings is achieved by NonInc with the use of a beam. These could be the reasons for NonInc performing better than RevInc in the case of a beam.

4.4.5 No Look-ahead

All the above experiments use a look-ahead for POS-tagging, supertagging and parsing. All the features involving input in Table 4.1 are look-ahead features. In this section, we analyze the impact of look-ahead for our RevInc algorithm. We re-trained POS-tagger, supertagger and parser without a look-ahead. On the CCGbank development data, the accuracies of POS-tagger, supertagger and multitagger with a look-ahead are 96.11%, 91.83% and 98.11% respectively. Without look-ahead the accuracies are 95.20%, 80.11% and 98.12% respectively. Look-ahead has a slight impact on the performance of the POS-tagger. But removing the look-ahead dropped the accuracy of the supertagger drastically by around 12%. However, the accuracy of the multitagger is almost the same with or without look-ahead. Since the parser takes the

Algorithm	*Beam*	*Look-ahead*	*UP*	*UR*	*UF*	*LP*	*LR*	*LF*	*Cat Acc.*
NonInc	1	Yes	**92.57**	82.60	87.30	**85.12**	75.96	80.28	**91.10**
RevInc	1	Yes	91.62	**85.94**	**88.69**	83.42	**78.25**	**80.75**	90.87
NonInc	16	Yes	**92.71**	**89.66**	**91.16**	**85.78**	**82.96**	**84.35**	**92.51**
RevInc	16	Yes	91.08	87.70	89.36	83.51	80.40	81.92	91.24
RevInc	1	No	81.39	56.98	67.03	68.93	48.26	56.77	74.79
RevInc	16	No	**90.18**	**85.96**	**88.02**	**82.27**	**78.42**	**80.30**	**89.64**
Z&C*	16	Yes	-	-	-	87.43	83.61	85.48	93.12

Table 4.3: Performance on the development data. *: These results are from Zhang and Clark (2011a).

output from the multitagger there is not a huge difference in the input to the parser in both scenarios.

The third block (5-6 rows) in Table 4.3 presents the results without look-ahead. As expected, removing the look-ahead significantly reduced the parsing accuracy. Interestingly, in the case of beam, the results without look-ahead (LF: 80.84%) are only 1.0% lower than the ones using a look-ahead (LF: 81.93) and similar to the greedy results with a look-ahead (LF: 80.75). For applications like web-scale parsing we can use the greedy RevInc parser with a look-ahead as it gives a nice trade-off between speed and accuracy. But for psycholinguistic experiments where strictly incremental nature is required, we can use the RevInc parser with a beam and no look-ahead.

4.4.6 Final Test Results

Table 4.4 presents the results of our approaches on the test data. We observed similar results as that on the development data. In greedy settings (first block), our incremental algorithm, RevInc, gives 2.0% and 0.88% improvements over NonInc in unlabelled and labelled F-scores respectively on the test data. Whereas in case of beam (second block), results with NonInc are slightly better than RevInc. Removing the look-ahead significantly reduced the accuracy but made the parser more cognitively plausible. With our greedy incremental parser which uses a look-ahead, we obtained a labelled F-score of 81.43%. We achieved a labelled F-score of 80.84% with our incremental parser with a beam and no look-ahead.

We compare our results with the incremental models of Hassan et al. (2009) (Hassan* in Table 4.4). They reported unlabelled F-scores of 86.31% and 59.01% with and without look-ahead respectively on test data which are 2.69% and 8.5% lower than the results with our greedy RevInc parser with and without look-ahead respectively. Note that these F-scores are not directly comparable since Hassan et al. (2009) use simplified lexicalized CCG categories. Our evaluation is based on CCG dependencies which are different from dependencies in the dependency grammar. Hence, we can't directly compare our results with dependency parsers like Zhang and Nivre (2011) and Honnibal et al. (2013). For unlabelled parsing we have to find the correct head for the word and for labelled parsing we have to find correct head as well as correct label. In the case of dependency parsing the labels are grammatical roles like `SUBJ` for subject or `OBJ` for object. But for CCG parsing, the CCG category and the index is considered as the label. Example dependencies for both CCG and dependency grammar formalisms

Algorithm	*Beam*	*Look-ahead*	*UP*	*UR*	*UF*	*LP*	*LR*	*LF*	*Cat Acc.*
NonInc	1	Yes	**92.45**	82.16	87.00	**85.59**	76.06	80.55	91.39
RevInc	1	Yes	91.83	**86.35**	**89.00**	84.02	**79.00**	**81.43**	91.17
NonInc	16	Yes	**92.68**	**89.57**	**91.10**	**86.20**	**83.32**	**84.74**	**92.70**
RevInc	16	Yes	91.22	88.46	89.82	84.06	81.52	82.77	92.11
RevInc	1	No	82.18	57.29	67.51	69.96	48.77	57.47	74.55
RevInc	16	No	**90.23**	**86.50**	**88.33**	**82.59**	**79.17**	**80.84**	**90.77**
Z&C*	16	Yes	-	-	-	87.43	83.61	85.48	93.12
Hassan*	1	Yes	-	-	86.31	-	-	-	-
Hassan*	1	No	-	-	59.01	-	-	-	-

Table 4.4: Performance on the test data. *: These results are from Zhang and Clark (2011a) and Hassan et al. (2009).

are provided below.

CCG:

$$< likes, (S\backslash NP_1)/NP_2, 1, John >$$

$$< likes, (S\backslash NP_1)/NP_2, 2, mangoes >$$

Dependency:

$$< likes, SUBJ, John >$$

$$< likes, OBJ, mangoes >$$

For a dependency between `likes` and `John`, a dependency parser has to identify the `SUBJ` label. But a CCG parser has to assign both the correct CCG category and the correct index, here *(S\NP)/NP* and 1 respectively.

4.4.7 Label-wise Impact

We analyzed the label-wise scores of both NonInc and RevInc in the greedy settings. In general, NonInc is better in precision and RevInc is better in recall. In the case of verbal arguments (*(S\NP)/NP*) and verbal modifiers (*(S\NP)\(S\NP)*), the F-score of RevInc is better than that of NonInc. But NonInc performed better than RevInc in the case of prepositional phrase (PP) attachments (*(NP\NP)/NP*, *((S\NP)\(S\NP))/NP*). More context is required for better PP attachment which is provided by the fact that NonInc has a context of several unreduced types for the model to work with, whereas RevInc has fewer. On the other hand, actions like `LRev` are required to correctly attach the verbal modifiers ('madly') if the subject argument ('John') of the verb ('likes') is reduced early. Table 4.5 gives the results of these CCG lexical categories.

4.4.8 Speed

We also analyzed the performance of the greedy (beam=1) NonInc and RevInc parsers in terms of parsing speed (excluding POS-tagger and supertagger time). Table 4.6 presents these results. NonInc and RevInc parse 110 and 125 sentences/second respectively. Despite the complexity of the revealing actions, RevInc is faster than NonInc. A significant amount of parsing time is spent on the feature extraction step. Features from the top four nodes in the stack and their children are extracted for both the algorithms. Since the average connectedness of RevInc and NonInc are 4.62 and 2.15 respectively, on average, all four nodes in the stack are processed for NonInc and only two nodes are processed for RevInc. Because of this there is significant reduction in

Category	*RevInc*	*NonInc*
*(NP\NP)/**NP***	81.36	**83.21**
*(NP**NP**)/NP*	78.66	**82.94**
*((S\NP)\(S\NP))/**NP***	65.09	**66.98**
*((S\NP)**(S\NP)**)/NP*	62.69	**65.89**
*((S[dcl]\NP)/**NP***	**78.96**	78.29
*((S[dcl]**NP**)/NP*	**76.71**	75.22
*(S\NP)**(S\NP)***	**80.49**	76.90

Table 4.5: Label-wise F-score of RevInc and NonInc parsers (both with beam=1). Argument slots in the relation are in bold.

the feature extraction step for RevInc compared to NonInc. Also, the complex LRev and RRev actions only constituted 5% of the total actions in the parsing process.

Model	*Sentences/Second*
NonInc	110
RevInc	125

Table 4.6: Speed comparison of NonInc and RevInc algorithms.

4.5 Conclusion

We have designed and implemented a new incremental shift-reduce algorithm based on a version of revealing for parsing CCG (Pareschi and Steedman, 1987). On the standard CCGbank test data, our algorithm achieved improvements of 0.88% and 2.0% in labelled and unlabelled F-scores respectively in the greedy settings over the baseline non-incremental shift-reduce algorithm. We also analyzed the impact of beam and look-ahead and showed that we can achieve comparable accuracies without a look-ahead when we use a beam. We achieved this without changing any CCG lexical categories and only changing a single head rule of making the main verb rather than the auxiliary verb the head. Our algorithm models transitions rather than incremental derivations, and hence we do not need an incremental CCGbank. Our approach can therefore be adapted to languages with dependency treebanks, since CCG lexical categories can be easily extracted from dependency treebanks (Cakici, 2005; Ambati

et al., 2013). We also designed new measures of incrementality and showed that our algorithm is more incremental than the standard shift-reduce CCG parsing algorithm.

We presented two versions of incremental parsers which gave accuracies comparable to state-of-the-art shift-reduce CCG parsers. The first one is a greedy parser which uses a look-ahead. The second one is a beam-search parser which does not use a look-ahead. In Chapter 6 we show how these incremental parsers can be useful in assessing relative sentence complexity. In the current chapter we worked with English which is a SVO language. In the next chapter we extend this algorithm to handle Hindi, a verb final language.

Chapter 5

Incremental Parsing for Hindi

In this chapter, we present transition-based CCG parsers for Hindi. We first extend Zhang and Clark (2011a)'s shift-reduce model by adding Hindi specific features to build the first shift-reduce CCG parser for Hindi. We analyze the impact of different settings of the parser, like chunk and morphological features, greedy vs. beam-search parsing, gold vs. automatic features, and coarse-grained vs. fine-grained lexicon. With automatic features, the beam-search parser with a coarse-grained lexicon gave best unlabelled and labelled F-scores of 85.60% and 77.32% respectively. We then design an incremental algorithm extending the revealing based incremental algorithm presented in the previous chapter. We make several extensions to make the algorithm as incremental as possible.

5.1 Introduction

Using the Hindi CCGbank described in chapter 2, we develop different transition-based CCG parsers for Hindi in this chapter. Ours is the first CCG parser for Hindi and the first transition-based parser for a non-English language.

We first present a shift-reduce CCG parser for Hindi. We extend Zhang and Clark (2011a)'s shift-reduce model by adding Hindi specific features. In that process, we also develop Hindi tools like a POS-tagger, chunker and supertagger. We observe the usefulness of different lexical and morphological features for developing a Hindi supertagger. Then we build the first shift-reduce CCG parser for Hindi. We analyze the impact of chunk and morphological features, greedy vs. beam-search parsing, gold vs. automatic features, and a coarse-grained vs. fine-grained lexicon. With automatic features, the beam-search parser with coarse-grained lexicon gave the best unlabelled

and labelled F-scores of 85.60% and 77.32% respectively.

Hindi, though predominantly having an Subject-Object-Verb (SOV) word order, is a free word order language and is morphologically rich. These characteristics make the task of incremental parsing extremely challenging for Hindi. We design an incremental algorithm extending the revealing based incremental algorithm presented in the previous chapter. We make several extensions to make the algorithm as incremental as possible. On the final test set, we obtained much lower accuracies with the incremental algorithm, as compared to the non-incremental one. Free word order, ambiguity in the morphological markers and low accuracy of the supertagger are some of the reasons for this. We provide a detailed discussion about the reasons for the low accuracy.

The rest of the chapter is arranged as follows. Section 5.2 gives a brief introduction to related work in the areas of CCG parsing for non-English languages and Hindi dependency parsing. We describe the non-incremental and incremental algorithms in section 5.3. In section 5.4, we provide details about different tools developed. Since ours is the first CCG parser for Hindi, we first present the details of different experiments conducted in section 5.5. Then, in section 5.6 we present the incremental parsing experiments. We conclude with possible future directions in section 5.7.

5.2 Related Work

We provided related work in the areas of CCG parsing, incremental and greedy parsing in the previous chapter. In this section, we provide details about CCG parsing for non-English languages and Hindi dependency parsing.

5.2.1 CCG Parsing

Though most of the literature is on parsing English CCGbank, there is some work on CCG parsing for non-English languages. Tse and Curran (2012) trained two English CCGbank parsers: the parser of Petrov and Klein (2007), and Clark and Curran (2007)'s C&C parser on Chinese CCGbank data. They obtained a labelled F-score of 72.73% with Petrov and Klein (2007)'s parser and 67.09% with the C&C parser. They also analyzed the challenges involved in parsing Chinese using CCG. Uematsu et al. (2013) developed a CCGbank for Japanese and trained Miyao and Tsujii (2008)'s Head-driven phrase structure grammar (HPSG) parsing model on a Japanese CCGbank.

For both Chinese and Japanese CCGbanks, state-of-the-art CCG parsers (Clark and Curran, 2007) and/or phrase structure parsers (Petrov and Klein, 2007; Miyao and Tsujii, 2008) are used to train a parsing model. These parsers require CCGbanks in order to be trained. But Cakici (2005) only developed a CCG lexicon for Turkish and these parsers could not be used to build a CCG parser for Turkish. Hence, she used Clark and Curran (2006)'s partial training approach to build a CCG parser for Turkish. Clark and Curran (2006)'s approach doesn't need a CCGbank. They take a CCG lexicon as input and generate a CCG derivation using CCG combinators. This approach is really useful for languages or domains where there are CCG lexicons available but not CCGbanks.

5.2.2 Hindi Dependency Parsing

Following the CoNLL shared tasks on dependency parsing (Buchholz and Marsi, 2006; Nivre et al., 2007a), two ICON Tools Contests (Husain, 2009; Husain et al., 2010) and a Hindi parsing shared task in Coling 2012 workshop on Machine Translation and Parsing in Indian Languages (Sharma et al., 2012) were held aiming at developing dependency parsers for three Indian languages, namely Hindi, Telugu and Bangla. Different rule-based, constraint based, statistical and hybrid systems were explored.

A two-stage approach for parsing Hindi is presented by Bharati et al. (2009a). They handle simple sentences in the first stage and then handle complex sentences in the second stage. Integer Linear Programming (ILP) technique is used to solve the constraint optimisation problem. Bharati et al. (2008) presented the first statistical parser for Hindi. They trained Malt and MST parsers on the Hindi dependency treebank. They explored the usefulness of different syntactic, morphological and semantic features for parsing Hindi. Since Hindi is a morphologically rich language, morphological features played a crucial role in identifying the dependency relations. Ambati et al. (2010a) and Ambati et al. (2010b) experimented with different morphological features and different methods for incorporating these features for Hindi dependency parsing. They experimented with both gold features and automatic features and showed that for Hindi, gender, number and person features are useful only in the gold settings. Whereas, postposition markers for nouns and Tense Aspect Modality (TAM) markers for verbs are shown to be useful in both the settings. Gadde et al. (2010) showed that using chunk information in addition to other syntactic and morphological features gave slight improvements in accuracy.

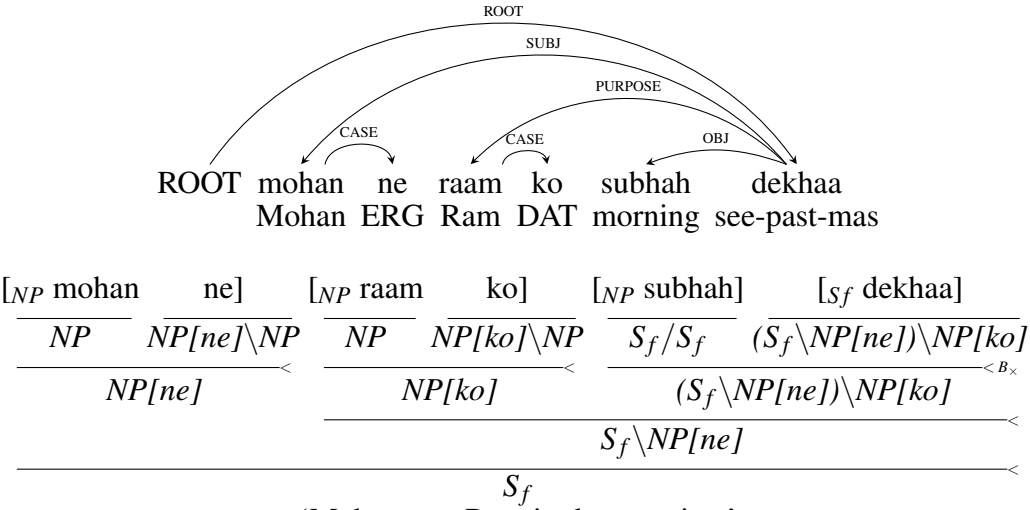


Figure 5.1: An example Hindi sentence with dependency tree and corresponding CCG derivation.

5.3 Algorithms

In this section, we first present our non incremental shift-reduce algorithm for parsing Hindi. Then we describe the extensions made to the revealing based incremental algorithm to handle Hindi.

5.3.1 Non Incremental Algorithm (NonInc)

This algorithm is based on Zhang and Clark (2011a) and is similar to the NonInc algorithm presented in the previous chapter for English. It consists of an input buffer and a stack and has four major parsing actions: `Shift`, `Reduce-Left`, `Reduce-Right` and `Reduce Unary`, which are described below in detail with an example sentence.

Figure 5.1 shows a normal-form CCG derivation for an example sentence 'mohan ne raam ko subhah dekhaa (Mohan saw Ram in the morning)'. Figure 5.2 shows the sequence of steps for parsing the sentence using the NonInc algorithm. For simplicity, we represent S_f as S without the finite feature. In steps 1 and 2, we shift the first two words in the sentence, 'mohan (Mohan)' and 'ne (ERG)', with corresponding CCG categories. Since 'ne (ERG)' is the case marker for the head noun 'mohan (Mohan)' these two nodes are combined using Reduce-Right (RR) action to form *NP[ne]:mohan*. Steps 4 and 5 shift the next two words 'raam (Ram)' and 'ko (DAT)' with *NP* and *NP[ko]\NP* categories respectively. Similar to the 'mohan ne' noun

(1)	S	[	NP_{mohan}		
(2)	S	[	NP_{mohan}	$NP[ne]\backslash NP_{ne}$	
(3)	RR	[	$NP[ne]_{mohan}$		
(4)	S	[	$NP[ne]_{mohan}$	NP_{raam}	
(5)	S	[	$NP[ne]_{mohan}$	NP_{raam}	$NP[ko]\backslash NP_{ko}$
(6)	RR	[	$NP[ne]_{mohan}$	$NP[ko]_{raam}$	
(7)	S	[	$NP[ne]_{mohan}$	$NP[ko]_{raam}$	S/S_{subhah}
(8)	S	[	$NP[ne]_{mohan}$	$NP[ko]_{raam}$	S/S_{subhah} $(S\backslash NP[ne])\backslash NP[ko]_{dekhaa}$
(9)	RL	[	$NP[ne]_{mohan}$	$NP[ko]_{raam}$	$(S\backslash NP[ne])\backslash NP[ko]_{dekhaa}$
(10)	RL	[	$NP[ne]_{mohan}$	$S\backslash NP[ne]_{dekhaa}$	
(11)	RL	[	S_{dekhaa}		

Figure 5.2: Sequence of actions with parser configuration for Hindi NonInc parser.

phrase, these two nodes are combined using RR action to form *NP[ko]:raam*. Next the adjunct 'subhah (morning)' is shifted followed by the main verb 'dekhaa (saw)' in steps 7 and 8. In step 9, *(S\NP[ne])\NP[ko]:dekhaa* is combined with *S/S:subhah* to form *(S\NP[ne])\NP[ko]:dekhaa*. Since 'subhah (morning)' is an adjunct, the result category is same as the category of the main verb 'dekhaa (saw)'. In step 10 *(S\NP[ne])\NP[ko]:dekhaa* is combined with *NP[ko]:raam* using Reduce-Left (RL) action leading to *S\NP[ne]:dekhaa*. This action resolves the object argument of the main verb. Then, *S\NP[ne]:dekhaa* is combined with *NP[ne]:mohan* to form *S:dekhaa* using RL action. This action resolves the subject argument. The parsing process terminates at this step as there are no more tokens in the input buffer and there is only a single node left in the stack.

We presented the algorithm using fine-grained CCG categories. The algorithm works the same for coarse-grained categories as well. In the coarse-grained lexicon we don't have morphological markers on categories. For example, the category of 'dekhaa (saw)' will be *(S\NP)\NP* without any case markers for *NP*s. Similar to English, we use indexed CCG categories and obtain CCG dependencies for evaluation.

5.3.2 Revealing based Incremental Algorithm (RevInc)

Since Hindi is predominantly a verb final language, as is seen in the above example, we have to wait till the end of the sentence to resolve arguments like subject and object. In this section, we introduce a new incremental algorithm extending the incremental algorithm for English described in 4.3.2 to handle verb final languages like Hindi. We take advantage of type-raising and type-changing rules in achieving this.

Figure 5.3 presents the incremental derivation for the example sentence discussed in the previous section. Figure 5.4 gives the steps involved in getting this derivation. We shift the first two words 'mohan (Mohan)' and 'ne (ERG)' with their corresponding CCG categories in steps 1 and 2. Step 3 combines these two nodes using the Reduce-Right (RR) action leading to *NP[ne]:mohan*. The first three steps are similar to the case of the NonInc algorithm. In NonInc, the next node is shifted to the stack. But in the incremental algorithm we type-raise *NP[ne]* to *S/(S\NP[ne])* using the Reduce-Unary (RU) action in step 4. As a result, the subject noun phrase 'mohan ne (Mohan ERG)' will demand a verb looking for a subject argument. In steps 5 and 6, the next two words, 'raam (Ram)' and 'ko (DAT)', are shifted with their categories. Step 7 combines these two nodes with the RR action and results in *NP[ko]:raam*

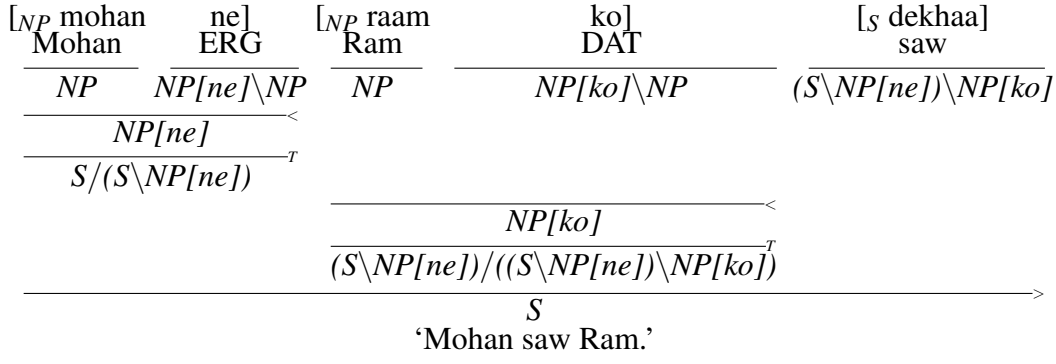
Figure 5.3: Incremental CCG derivation for an example Hindi sentence.

(1)	S	[	NP_{mohan}		
(2)	S	[	NP_{mohan}	$NP[ne]\backslash NP_{ne}$	
(3)	RR	[	NP_{mohan}		
(4)	RU	[	$S/(S\backslash NP[ne])_{mohan}$		
(5)	S	[	$S/(S\backslash NP[ne])_{mohan}$	NP_{raam}	
(6)	S	[	$S/(S\backslash NP[ne])_{mohan}$	NP_{raam}	$NP[ko]\backslash NP_{ko}$
(7)	RR	[	$S/(S\backslash NP[ne])_{mohan}$	$NP[ko]_{raam}$	
(8)	RU	[	$S/(S\backslash NP[ne])_{mohan}$	$(S\backslash NP[ne])/((S\backslash NP[ne])\backslash NP[ko])_{raam}$	
(9)	RL	[	$S/((S\backslash NP[ne])\backslash NP[ko])_{raam}$		
(10)	S	[	$S/((S\backslash NP[ne])\backslash NP[ko])_{raam}$	S/S_{subhah}	
(11)	RU	[	$S/((S\backslash NP[ne])\backslash NP[ko])_{raam}$	$((S\backslash NP)\backslash NP)/((S\backslash NP)\backslash NP)_{subhah}$	
(12)	RL	[	$S/((S\backslash NP[ne])\backslash NP[ko])_{subhah}$		
(13)	S	[	$S/((S\backslash NP[ne])\backslash NP[ko])_{subhah}$	$(S\backslash NP[ne])\backslash NP[ko]_{dekhaa}$	
(14)	RL	[	S_{dekhaa}		

Figure 5.4: Sequence of actions with parser configuration for Hindi RevInc parser.

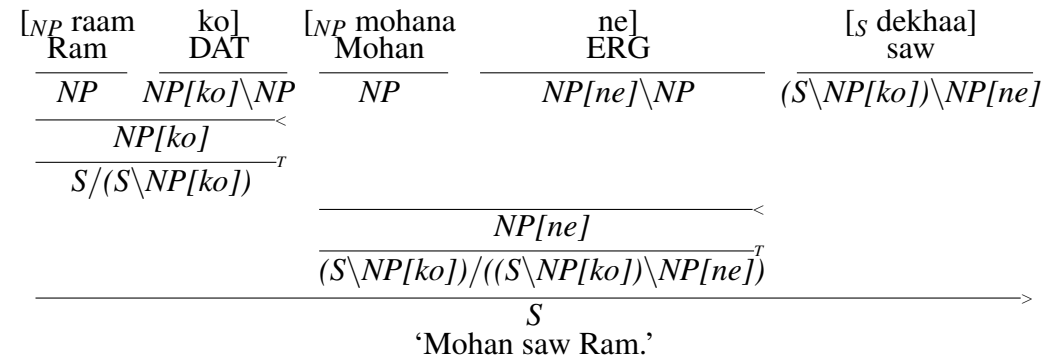
node. Similar to the subject noun phrase, this object noun phrase is also type-raised to *(S\NP[ne])/((S\NP[ne])\NP[ne]:mohan* in step 8. Now, both the subject and object arguments are combined in step 9 to form a node *S/((S\NP[ne])\NP[ko]* which demands a transitive verb to its right. Since there is no dependency between these two nodes we consider the right node as the head for the purposes of the derivation. In steps 10 and 11, the adjunct 'subhah (morning)' is shifted and type-changed to *((S\NP) \NP)/((S\NP)\NP)* so that it can combine with the argument cluster 'mohan ne raam ko'. In step 12, both the argument cluster and the adjunct are combined to form *S/((S\NP[ne])\NP[ko]*. Step 13 shifts the main verb 'dekhaa (saw)' with its category *((S\NP[ne])\NP[ko])* which is combined with *S/((S\NP[ne])\NP[ko]* leading to *S* in step 14.

Since Hindi is a free word order language, handling different word orders with the incremental analysis is tricky. Consider a simpler version of the above example sentence with just subject, object and verb information in SOV and OSV orderings. 'mohan ne raam ko dekhaa (Mohan saw Ram)' and 'raam ko mohan ne dekhaa (Mohan saw Ram)' are the sentences in SOV and OSV word order respectively. Figure 5.5 presents the incremental derivations for both these sentences. If you look at the subject noun phrase 'mohan ne (Mohan ERG)', its category *NP[ne]* is type-raised to *S/(S\NP[ne])* in the SOV case and *(S\NP[ko])/((S\NP[ko])\NP[ne])* in the OSV case. Things get more complicated when different nodes like adjuncts are added to the sentence. In this case, we are keeping track of the previous node's categories while type-raising the current node. Instead we can solve this by considering the information only from the current node when type-raising. 'mohan ne (Mohan ERG)' just gives us the information that it demands a verb which looks for a noun in ergative case. It doesn't give any information about the number of other arguments or their ordering. So, we can just type-raise the category to demand a verb looking for an ergative case marked noun. We can further type-change the category based on the position of the current node with respective to other nodes. Figure 5.6 shows the incremental derivations using this idea. Here, in both SOV and OSV sentences, 'mohan ne (Mohan ERG)' is type-raised to *S/(S\NP[ne])*. In SOV ordered sentence, this category is not changed. But in OSV ordered sentence, this is further type-changed to *(S\NP[ko])/((S\NP[ko])\NP[ne])* based on the context. This small change makes the task of the parser simpler as it avoids the category explosion caused by different word orderings.



'Mohan saw Ram.'

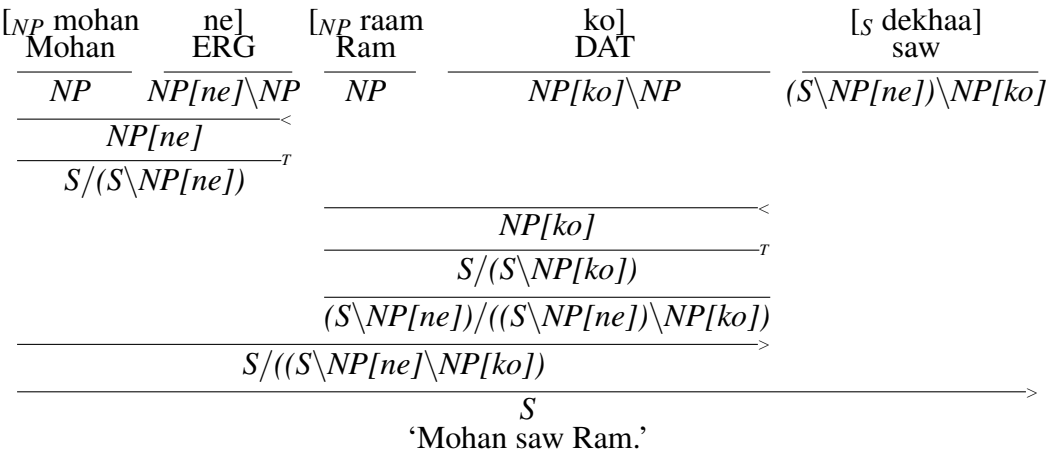
(a) SOV word order



'Mohan saw Ram.'

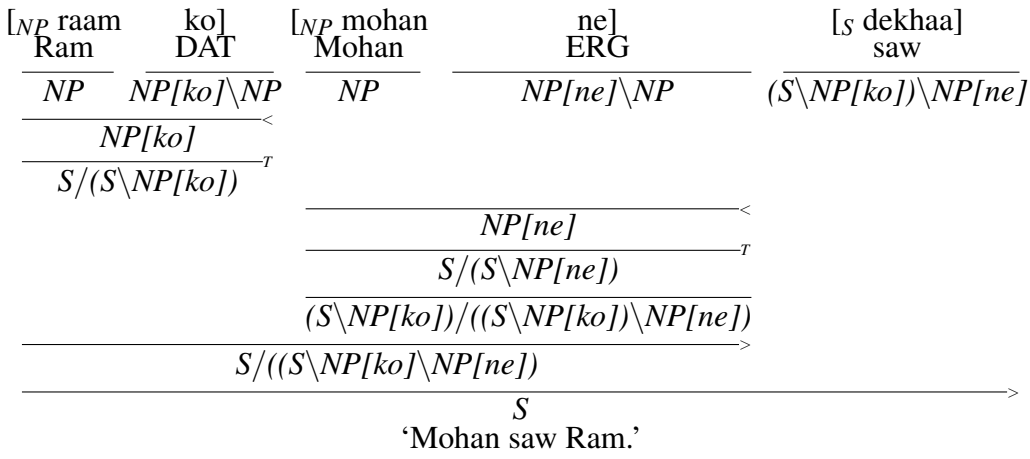
(b) OSV word order

Figure 5.5: Derivations for SOV and OSV word orderings with single type-raising.



‘Mohan saw Ram.’

(a) SOV word order



‘Mohan saw Ram.’

(b) OSV word order

Figure 5.6: Derivations for SOV and OSV word orderings with type-raising followed by a type-changing rule.

5.4 Tools and Settings

5.4.1 Shallow Parser

The shallow parser for Hindi[1] consists of a morphological analyzer, Part-Of-Speech (POS) tagger and a chunker. The morphological analyzer gives morphological features like the root form of the word, gender (masculine/feminine/neutral), number (singular/plural), person (first/second/third), case (oblique/direct), pos-position markers or tense, aspect and modality (TAM) markers and suffix. The POS tagger and chunker assign POS tags and chunk tags respectively. For example, in the sentence in Figure 5.7, 'mohan ne raam ke_lie kitaab khariidii (Mohan bought a book for Ram)', 'mohan ne (Mohan)', 'raam ke_lie (for Ram)' and 'kitaab (book)' are *NP* chunks and 'khariidii (bought)' is a finite verb chunk represented as S_f.

[$_{NP}$ mohan	ne]	[$_{NP}$ raam	ke_lie]	[$_{NP}$ kitaab]	[$_{Sf}$ khariidii]
Mohan	ERG	Ram	for	book	buy-past-fem

'Mohan bought a book for Ram.'

Figure 5.7: An example Hindi sentence with chunk information.

The POS and chunk tagsets used in the Hindi dependency treebank are slightly different from the tags provided by the shallow parser, as the Hindi shallow parser uses an older tagset. For example, in the older tagset finite, non-finite, infinite information about the verbs is present at the POS level with the use of tags like VFM (fininte), VNN (non-finite) etc. Whereas in the new tagset, all the non-auxiliary verbs get the same tag (VM) and finite/non-finite information is moved to the chunk level represented with VGF, VGNF tags[2]. To avoid this inconsistency, we trained a maximum entropy based POS tagger and chunker on the Hindi dependency treebank. We used training data for training the model, and development data to tune the model. We used the same set of features used to build the taggers in the shallow parser (Avinesh and Gali, 2007). For POS tagging, lexical items like the current word, the previous two words and the next two words are used as features. In addition to the current word, suffixes and prefixes of lengths 1-4 for the current word are also provided as features. For the chunker, features from word and POS tags of the current word, previous two words and next two words are used to build the model. Performance of the POS tagger on the development and

[1] http://ltrc.iiit.ac.in/analyzer/hindi/

[2] See Appendix A.1 and A.2 for complete list.

Experiments: Features	*Accuracy*	
	Fine-grained	*Coarse-grained*
Exp 1: w_i, p_i	75.14	78.47
Exp 2: *Exp 1* + l_i, t_i	77.58	80.17
Exp 3: *Exp 2* + f_i	80.43	81.88
Exp 4: *Exp 3* +w_{i-1}, w_{i-2}, p_{i-1}, p_{i-2}, w_{i+1}, w_{i+2}, p_{i+1}, p_{i+2}	82.72	84.15
Exp 5: *Exp 4* + w_ip_i, w_it_i, w_if_i, p_if_i	82.81	84.29
Exp 6: *Exp 5* + $w_{i-2}w_{i-1}$, $w_{i-1}w_i$, w_iw_{i+1}, $w_{i+1}w_{i+2}$, $p_{i-2}p_{i-1}$, $p_{i-1}p_i$, p_ip_{i+1}, $p_{i+1}p_{i+2}$	**82.92**	**84.40**

Table 5.1: Impact of different features on the supertagger performance.

test data respectively is 93.08% and 93.16%. With gold POS tags, the chunker gave an accuracy of 97.35% and 97.26% on development and test data respectively.

5.4.2 Supertagger

Clark (2002) developed the first CCG supertagger for English. A maximum entropy model was used to build this supertagger. Curran and Clark (2003) explored different options like Generalised Iterative Scaling (GIS) estimation and model smoothing to improve the supertagger. Extending the supertagger model, Clark (2002) built a multitagger for English. A multitagger assigns all categories to a word whose probabilities are within some factor, β, of the category with the highest probability. It has been shown that using multitagger output instead of supertagger output while parsing gives better performance (Clark and Curran, 2004b). The state-of-the-art CCG parsers for English (Clark and Curran, 2007) use multitagger output during parsing.

Following Clark (2002), we used maximum entropy models to build supertaggers and multitaggers for Hindi. Details of the features used and other settings for the Hindi

supertagger are provided in the following sections.

5.4.2.1 Category Set

For supertagging, we first obtained a category set from the Hindi CCGbank training data. There are 2,177 and 718 category types in fine-grained (with morph. information) and coarse-grained (without morph. information) data respectively. Clark and Curran (2004b) showed that using a frequency cutoff can significantly reduce the size of the category set with only a small loss in coverage. We explored different cut-off values and finally used a cutoff of 10 for building the tagger. This reduced the category types to 376 and 202 for fine-grained and coarse-grained lexicon respectively. The percent of category tokens in the development data that don't appear in the category set with this cut-off are 1.39 & 0.47 for the fine-grained and coarse-grained lexicon respectively.

5.4.2.2 Features

Following Clark and Curran (2004b), we used a Maximum Entropy approach to build our supertagger. We explored different features in the context of a 5-word window surrounding the target word. We used features based on WORD (w), LEMMA (l), POS (p), CPOS TAG (t) and the FEATS (f) columns of the CoNLL format. Table 5.1 shows the impact of different features on the supertagger performance. Experiments 1, 2, 3 have current word (w_i) features while Experiments 4, 5, 6 show the impact of contextual and complex bi-gram features. $i-1$, $i-2$ represent previous two positions and $i+1$, $i+2$ represent the next two positions.

Accuracy of the supertagger after Experiment 6 is 82.92% and 84.40% for fine-grained and coarse-grained lexicon respectively. As the number of category types in the fine-grained lexicon (376) are much higher than in the coarse-grained one (202), it is not surprising that the performance of the supertagger is better for the coarse-grained lexicon as compared to the fine-grained one.

5.4.2.3 Multi-tagger

We explored different values of β to develop a multitagger for Hindi. β of 0.01 gives better performance and reasonable number of categories per word. A performance of 97.90% and 96.82% is achieved on the testing data for coarse-grained and fine-grained lexicon respectively. Respective average number of categories per word is 5.27 and 6.65.

Tagger	*Gold Features*		*Automatic Features*	
	Coarse	Fine	Coarse	Fine
supertagger	88.05	86.91	84.40	82.92
multi-tagger	98.42	97.50	97.90	96.82

Table 5.2: Performance of multi-tagger on the Hindi CCGbank.

5.5 Experiments and Results: NonInc

We first trained our re-implementation of Zhang and Clark (2011a)'s (Z&C*) model on the Hindi data. We used their global linear model trained with the averaged perceptron (Collins, 2002). We applied the early-update strategy of Collins and Roark (2004) while training. In this strategy, when we don't use a beam, decoding is stopped when the predicted action is different from the gold action, and weights are updated accordingly. Z&C* use a beam of size 16 for their experiments. We explore the impact of morphological features, greedy vs. beam, and gold vs. automatic features. We adapted Hockenmaier's scripts to extract CCG dependencies from the Hindi CCGbank derivations.

5.5.1 Data and Settings

We use the Hindi CCGbank developed in chapter 2 for our experiments. We use the training, development and testing splits used for dependency parsing experiments. We experiment with both fine-grained and coarse-grained lexicons and also see the impact of gold and automatic POS, and chunk features. Also, for lexical CCG categories, we use a multitagger. We use the POS-tagger, chunker and multitagger described in the previous section for our experiments. Following Z&C* and Xu et al. (2014), during training, we also provide the gold CCG lexical category to the list of CCG lexical categories for a word if it is not assigned by the supertagger.

5.5.2 Impact of Morphological Features

We use the feature set of Zhang and Clark (2011a) (Z&C) for our baseline experiment. Here, we use coarse-grained lexicon and gold POS, chunk features. First row

Features	*UF*	*LF*	*Cat Acc.*
Z&C*	84.06	73.78	85.58
Z&C* + Lemma, Chunk	84.22	74.19	85.98
Z&C* + Lemma, Chunk + Morph	**85.47**	**75.89**	**87.55**

Table 5.3: Impact of features on the Hindi CCG parser

of Table 5.3 gives these results. With the English feature set, we obtain an unlabelled F-score (UF), labelled F-score (LF) and Category Accuracy (Cat Acc.) of 84.06%, 73.78% and 85.58%. Then we provided lemma, chunk features and morphological features. Lemma and chunk features gave small gains of 0.2-0.4%. Morphological features gave significant boost of 1.3-1.7%. Chunk features include the chunk tag of the word and whether the word is head/child in the chunk. For example, in the noun chunk 'mohan ne (Mohan ERG)', NP is the chunk tag for both the words. 'mohan (Mohan)' is the head of the chunk and 'ne (ERG)' is the child in the chunk. We explored different morphological features provided in the treebank. Suffix and case/TAM markers proved to be very useful. But gender, number, person information didn't give any improvements. This is similar to the Hindi dependency parsing experiments in Ambati (2011). Table 5.3 presents the results of all these experiments.

Table 5.4 shows a list of all the feature templates used for Hindi CCG parsing. Lemma, Chunk and Morphological rows are Hindi specific features and the rest are features adapted from the English CCG parser. Let S0, S1, S2, S3 be top four nodes in the stack and Q0, Q1, Q2, Q3 be the next four nodes in the input. Let the left and right children of the top two nodes in the stack be represented by S0L, S0R, S1L, S1R and head and unary head for the top two nodes in the stack are represented by S0H, S0U, S1H, S1U. Let w, l, p, t, c, f be word, lemma, POS-tag, chunk tag, CCG category and morphological features of a node respectively. We use three morphological features namely, suffix of the word, case/TAM marker and whether the word is head or child of the chunk. The top four rows in the table comprises of 64 features used for English NonInc CCG parsing. Lemma and Chunk, Morphological rows shows the additional features specific to Hindi CCG parsing. In total we used 83 feature templates for Hindi CCG parsing.

Type	*Features*
Stack (Basic)	S0wp, S0c, S0pc, S0wc, S1wp, S1c, S1pc, S1wc, S2pc, S2wc, S3pc, S3wc,
Input (Basic)	Q0wp, Q1wp, Q2wp, Q3wp,
Stack (Children)	S0Lpc, S0Lwc, S0Rpc, S0Rwc, S0Upc, S0Uwc, S1lpc, S1lwc, S1Rpc, S1Rwc, S1Upc, S1Uwc,
Bigram (Basic)	S0wcS1wc, S0cS1w, S0wS1c, S0cS1c, S0wcQ0wp, S0cQ0wp, S0wcQ0p, S0cQ0p, S1wcQ0wp, S1cQ0wp, S1wcQ0p, S1cQ0p,
Trigram (Basic)	S0wcS1cQ0p, S0cS1wcQ0p, S0cS1cQ0wp, S0cS1cQ0p, S0pS1pQ0p, S0wcQ0pQ1p, S0cQ0wpQ1p, S0cQ0pQ1wp, S0cQ0pQ1p, S0pQ0pQ1p, S0wcS1cS2c, S0cS1wcS2c, S0cS1cS2wc, S0cS1cS2c, S0pS1pS2p,
Trigram	S0cS0HcS0Lc, S0cS0HcS0Rc, S1cS1HcS1Rc, S0cS0RcQ0p, S0cS0RcQ0w, S0cS0LcS1c, S0cS0LcS1w, S0cS1cS1Rc, S0wS1cS1Rc
Lemma and Chunk	S0l, S0t, S0lt, S0tc, S1l, S1t, S1lt, S1tc Q0lt, Q1lt
Morphological	S0f, Q0f, S0fQ0f

Table 5.4: Feature templates for Hindi CCG parser.

Model	*UP*	*UR*	*UF*	*LP*	*LR*	*LF*	*Cat Acc.*
Fine (beam=1)	84.11	82.41	83.25	74.44	72.93	73.68	86.53
Fine (beam=16)	**84.59**	**90.81**	**87.59**	**77.09**	**82.77**	**79.83**	**90.45**
Coarse (beam=1)	84.89	86.06	85.47	75.37	76.41	75.89	87.55
Coarse (beam=16)	**85.82**	**92.70**	**89.12**	**78.71**	**85.02**	**81.74**	**91.16**

Table 5.5: Performance on the Hindi CCG parser on the testing data.

Model	*Greedy*			*Beam = 16*		
	UF	*LF*	*Cat Acc.*	*UF*	*LF*	*Cat Acc.*
Fine	79.21	68.84	82.19	83.82	74.80	86.37
Coarse	**81.43**	**71.17**	**83.83**	**85.60**	**77.32**	**87.64**

Table 5.6: Performance on the Hindi CCG parser using automatic features.

5.5.3 Lexicon and Beam

In this section, we analyze the impact of both the lexicon, and a beam for Hindi CCG parsing. Table 5.5 presents the results for both fine-grained and coarse-grained lexicon in greedy and beam settings. UP, UR, UF are unlabelled precision, recall and F-score respectively. Similarly LP, LR, and LF represent labelled precision, recall and F-score respectively. Cat Acc. is the CCG category accuracy. All these results are with gold POS and chunk features.

In greedy settings, we obtain 83.25% in UF and 73.68% in LF using the fine-grained lexicon. With the coarse-grained lexicon we get UF and LF of 85.47% and 75.89% respectively. Using the coarse-grained lexicon gives around 2.2% better results than the fine-grained lexicon. Given that the fine-grained tagset is three times larger than the coarse-grained one and supertagger accuracy is 1.5% lower than for the coarse-grained lexicon, this result is not surprising.

Similar to Zhang and Clark (2011a), we use a beam of size 16. Using a beam gave approximately a 4% boost in UF and a 6% boost in LF for both the lexicons.

5.5.4 Automatic Features

In the previous section, we analyzed the impact of lexicon and beam using gold POS, chunk features. In this section, we replicate the experiments with automatic features. Table 5.6 presents the unlabelled F-score (UF), labelled F-score (LF) and category accuracy (Cat Acc) for all these experiments.

In greedy settings, we achieve an LF of 68.84% with the fine-grained lexicon and 71.17% with the coarse-grained lexicon. The coarse-grained lexicon has less number of tags and also, the supertagger on coarse-grained lexicon is 1.5% better than on the fine-grained lexicon. As a result we see a 2.33% improvement in accuracy using the coarse-grained lexicon. Similar to the experiments with gold features in the previous section, using a beam showed improvements of 4% in UF and 6% in LF. We obtain final best results of 85.60% in UF, 77.32% in LF, and 87.64% in Cat Acc. with a coarse-grained lexicon and using a beam of size 16. On the English CCGbank test set, we get accuracies of 91.38% in UF and 85.00% in LF with a beam of 16. Final Hindi results are 6-8% lower than the English results. But note that Hindi is much complex than English due to its free word order and morphological richness. Also, the Hindi dataset is nearly four times smaller than the English dataset.

In general, the coarse-grained lexicon gives better results than the fine-grained lexicon and using a beam gives significant boost over the greedy parser. Morphological features played an important role in improving the performance of both the supertagger and the parser.

5.6 Experiments and Results: RevInc

In this section, we present the results of our incremental algorithm for Hindi CCG parsing. Similar to English experiments, we show the impact of greedy vs. beam search, and look-ahead vs. no look-ahead for Hindi. We use the same feature set used in the previous section. Since the coarse-grained lexicon gave better results than the fine-grained lexicon, we work with the coarse-grained lexicon and automatic features in this section. All the parameter tuning is done on the development data and the settings which gave best result for development data are directly used on the test data. Table 5.7 presents the results of all these experiments.

Model	*Beam*	*Look-ahead*	*UP*	*UR*	*UF*	*LP*	*LR*	*LF*	*Cat Acc.*
NonInc	1	Yes	**81.03**	**81.83**	**81.43**	**70.82**	**71.52**	**71.17**	**83.83**
RevInc	1	Yes	79.81	65.16	71.75	68.88	56.24	61.92	77.26
NonInc	16	Yes	**82.51**	**88.92**	**85.60**	**74.53**	**80.32**	**77.32**	**87.64**
RevInc	16	Yes	80.04	73.86	76.83	69.56	64.19	66.77	81.65
RevInc	1	No	68.87	42.80	52.79	49.60	30.82	38.02	51.36
RevInc	16	No	**79.17**	**72.43**	**75.65**	**67.99**	**62.20**	**64.97**	**80.20**

Table 5.7: Performance of NonInc and RevInc algorithms on Hindi.

5.6.1 Greedy

The first block of Table 5.7 presents the results of the greedy parsers. With NonInc, we obtain an unlabelled F-score (UF) and labelled F-score (LF) of 81.43% and 71.17% respectively. With RevInc, we get UF and LF of 71.75% and 61.92%, which is around 10% lower than the non-incremental algorithm. Unlabelled and labelled precision of RevInc is slightly lower than the NonInc. But the respective recalls are around 15% lower which is the main reason for the drop in F-score. Also, category accuracy is 5% lower for RevInc compared to NonInc. We analyzed the output to better understand the reasons for this low accuracy with RevInc. The free word order nature, morphological complexity and low training data are some of the main reasons for this low performance. We provide complete details in section 5.6.4.

5.6.2 Beam

The second block (3-4 rows) in Table 4.3 shows these results with a beam of size 16. Following Zhang and Clark (2011a) and to compare the results with English experiments, we use a beam of size 16. F-scores improved by 5-6% when we use a beam. Similar to the greedy settings, performance of RevInc is 9-10% lower than the NonInc algorithm.

5.6.3 No Look-Ahead

Greedy and the beam search experiments described above use a look-ahead for pos-tagging, chunking, super-tagging and parsing. In this section, we analyze the impact of look-ahead for our RevInc algorithm. We re-trained the taggers (pos-tagger, chunker and supertagger) and the parser without a look-ahead. Performance of pos-tagger and chunker decreased only by 1% without a look-ahead. The accuracy of the supertagger is significantly reduced from 84.40% to 76.63% when we removed the look-ahead features. However, the accuracy of the multitagger is 96.63% which is just 1% lower than the performance with a look-ahead (97.90%).

The third block (5-6 rows) in Table 5.7 presents results without look-ahead. Removing the look-ahead significantly reduced the parsing accuracy. There is around 20% drop in unlabelled and labelled F-scores in the greedy settings. Similar to English, in case of a beam, there is only slight decrease in the accuracy as the multitagger accuracy is similar to the case of using a look-ahead.

5.6.4 Analysis

In this section we analyze the reason for the drop in the accuracy for the incremental algorithm compared to the non-incremental version. The main problem is ambiguity in the morphological markers. Some nouns don't have morphological markers and even in the cases where they are present, morphological markers are ambiguous. Consider the sentence (1a). Noun phrases `kitaab` (book) and `kal` (yesterday) both don't have morphological markers. Here, `kitaab` (book) is the direct object, a mandatory argument, whereas, `kal` is a time expression which is an adjunct. In (1a) morphological markers are missing for an argument and an adjunct. In Hindi, we can have sentences comprising of two argument noun phrases without morphological markers as in (2). In sentence (2), both `mohan` (Mohan) and `kitaab` (book) don't have morphological markers. `mohan` is the subject and `kitaab` is the direct object of the verb `deta hai` (gives).

(1) a. mohan ne raam ko kitaab kal dii
Mohan ERG Ram DAT book yesterday gave
"Mohan gave a book for Ram yesterday" (S-IO-DO-V)

b. [mohan ne] [kitaab] [raam ko] [kal] [dii]

(2) [mohan][raamko] [kitaab] [deta hai]
Mohan Ram DAT book give is
"Mohan gives a book for Ram" (S-IO-DO-V)

(2) [raam ko] [kitaab] [khariidna pada]
Ram DAT book had-to-buy
"Ram had to buy the book" (S-DO-V)

(3) [mohan ne] [raam ko] [subhah] [dekhaa]
Mohan ERG Ram DAT morning saw
"Mohan saw Ram in the morning"

(4) [mohan ne] [raam ko] [ora] [syam ko] [dekhaa]
Mohan ERG Ram DAT and Shyam DAT saw
"Mohan saw Ram and Shyam"

Case markers on the nouns, when they are present, can be ambiguous. Consider the `ko` marker in the (1a) and (2) example sentences above. In (1a), `raam ko` is an indirect object. Whereas in (2), the same noun phrase `raam ko` is the subject argument of the verb. The tense, aspect and modality (TAM) marker for the verbs helps in correct

identification of these arguments. TAM marker of na_pada in (2) helps in identifying that the raam ko noun phrase is the subject argument rather than the indirect object as in (1a) where the TAM marker is ii.

In case of NonInc, as we wait till the end of the sentence, we assign the correct category to the verb taking advantage of the TAM marker for verbs and case markers for nouns. But in the case of RevInc, we are predicting the category of the verb without information about its TAM marker. Impact of the morphological markers is made clear when we observe the labelled F-scores of both NonInc and RevInc in the initial iterations during training. Labelled F-scores of NonInc and RevInc after the first iteration are 62.39% and 59.22% respectively. The difference in F-score between RevInc and NonInc is 3% after the first iteration. But this difference becomes wider, to 10%, after the final iteration. For a given context, over the iterations, NonInc assigns more probability to single best action. Whereas, the probability mass is distributed more among the conflicting cases for RevInc. For example, for kitaab over iterations, NonInc assigns it a direct object category. Whereas, in the case of RevInc, the probability mass is distributed between the direct object and adjunct relations.

The free word order nature of Hindi makes the task more complicated. Consider sentences 1(a) and 1(b). Both the sentences mean the same but have different word orderings. In the first case, the CCG category of the ditransitive verb dii (gave) is *(((S\NP[ne])\NP[ko])\NP[0])*. But in the second case the category would be *(((S\NP[ne])\NP[0])\NP[ko])*, since the order of the arguments changed. In the case of NonInc, we wait till the end of the sentence to assign a category for the verb. So, we will have enough information about the ordering of the arguments to assign a correct category. But in case of RevInc, we predict the category of the verb before we even see the verb. So, the main problem here is the assumption about the ordering of the arguments. We can avoid this by changing the lexicon following Baldridge (2002) which are underspecified as to argument order. For example, the category of dii (gave) would be *S* {*NP[ne], NP[ko], NP[0]*}. This category gives a result category of *S* once all the three arguments are resolved irrespective of the ordering of the arguments.

Having a beam with different analysis would be helpful. But as the length of the sentence increases, the number of possible variations increases exponentially making the task of identifying the correct categories very difficult. Using CCG categories similar to Baldridge (2002) and using more training data might be useful. Using techniques like tri-training (Weiss et al., 2015), we can obtain more training data from the raw corpus.

Currently our revealing actions rely on the dependency graph created based on the dependencies resolved from the CCG derivation. Since Hindi is a verb final language, merging the argument cluster doesn't give any dependencies, since the verb is absent. For example, consider (4) which is the case of object coordination. Using type-raising rules, we can merge mohan and raam. For coordination we need to reveal raam which is currently not possible since we don't have that information in the dependency graph (as the graph is empty). So, we first merge the object arguments raam and syam and then combine them with the subject argument mohan to get a derivation. As a result, arguments are combined immediately when there is no coordination and in case of coordination we wait till the nodes in conjunction are combined. So, the probability mass is distributed between both the forms of derivations. We can make a minor change in the algorithm where we create a dependency graph with a dummy verb. When mohan and raam are combined, the dependency graph is updated with a dummy verb with mohan as its subject and raam as its object. Now we can use the right reveal action to reveal raam for a coordination construction, making the derivation more incremental.

5.7 Conclusion

We first presented transition-based CCG parsers for Hindi. We extended Zhang and Clark (2011a)'s shift-reduce model by adding Hindi specific features to build the first shift-reduce CCG parser for Hindi. We analyzed the impact of different settings like chunk and morphological features, greedy vs. beam-search parsing, gold vs. automatic features, and using a coarse-grained vs. fine-grained lexicon. With automatic features, the beam-search parser with coarse-grained lexicon gave the best unlabelled and labelled F-scores of 85.60% and 77.32% respectively. Then we designed an incremental algorithm extending the revealing based incremental algorithm presented in the previous chapter. We made several extensions to make the algorithm as incremental as possible. Unlike English, we got much lower results for Hindi using the incremental algorithm compared to the non-incremental version. We analyzed the reasons for this result for Hindi as arising from its free word order nature, combined with morphological ambiguity and low volumes of training data. We have suggested that the algorithm can be improved by using an underspecified lexicon similar to Baldridge (2002) and by introducing new actions for handling coordination.

Chapter 6

Assessing Relative Sentence Complexity using Incremental Parsers

In this chapter, we see how incremental CCG parsers can help in a practical application like assessing relative sentence complexity. Given a pair of sentences, we present computational models to assess if one sentence is simpler to read than the other. While existing models have explored the usage of phrase structure features using a non-incremental parser, experimental evidence suggests that the human language processor works incrementally. We empirically evaluate if syntactic features from incremental CCG parsers are more useful than features from a non-incremental phrase structure parser. Our evaluation on Simple and Standard Wikipedia sentence pairs shows that incremental CCG parser gives significant improvements in speed (12 times faster) as well as in terms of accuracy (0.44 points better) in comparison to the previously used Stanford parser. Furthermore, with the addition of psycholinguistic features, we achieve the strongest result to date reported on this task. Part of this work is published in Ambati et al. (2016c).

6.1 Introduction

The task of assessing text readability aims to classify text into different levels of difficulty, e.g., text comprehensible by a particular age group or second language learners (Petersen and Ostendorf, 2009; Feng, 2010; Vajjala and Meurers, 2014). There have been efforts to automatically simplify Wikipedia to cater its content for children and English language learners (Zhu et al., 2010; Woodsend and Lapata, 2011; Coster and Kauchak, 2011; Wubben et al., 2012; Siddharthan and Mandya, 2014). A re-

lated attempt of Vajjala (2015) studied the usage of linguistic features for automatic classification of a pair of sentences – one from Standard Wikipedia and the other its corresponding simplification from Simple Wikipedia – into COMPLEX and SIMPLE. As syntactic features, they use information from phrase structure trees produced by a non-incremental parser, and found them useful.

However, psycholinguistic theories suggest that humans process text incrementally, i.e., humans build syntactic analysis interactively by enhancing current analysis or choosing an alternative analysis on the basis of the plausibility with respect to context (Marslen-Wilson, 1973; Altmann and Steedman, 1988; Tanenhaus et al., 1995). Besides being cognitively possible, incremental parsing has shown to be useful for many real-time applications such as language modeling for speech recognition (Chelba and Jelinek, 2000; Roark, 2001), modeling text reading time (Demberg and Keller, 2008), dialogue systems (Stoness et al., 2004) and machine translation (Schwartz et al., 2011). Furthermore, incremental parsers offer linear time speed. Here we explore the usefulness of incremental parsing for predicting relative sentence readability.

Given a pair of sentences – one sentence a simplified version of the other – we aim to classify the sentences into SIMPLE or COMPLEX. We use the sentences from Standard Wikipedia (WIKI) paired with their corresponding simplifications in Simple Wikipedia (SIMPLEWIKI) as training and evaluation data. We pose this problem as a pairwise classification problem (Section 6.2). For feature extraction, we use an incremental CCG parser which provides a trace of each step of the parse derivation (Section 6.3). Our evaluation results show that incremental parse features are more useful than non-incremental parse features (Section 6.5). With the addition of psycholinguistic features, we attain the best reported results on this task.

6.2 Problem Formulation

Initially Vajjala and Meurers (2014) trained a binary classifier to classify sentences in SIMPLEWIKI to the class SIMPLE, and sentences in WIKI to the class COMPLEX. This model performed poorly on relative readability assessment mainly because each sentence is classified independently rather than treating SIMPLEWIKI and WIKI sentences as a pair. Table 6.1 presents two examples from OneStopEnglish[1] (OSE) corpus (Vajjala, 2015). This corpus consists of articles rewritten at three different reading levels (elementary, intermediate and advanced). Consider the intermediate sentence in the

[1] http://www.onestopenglish.com/

Advanced:	In Beijing, mourners and admirers made their way to lay flowers and light candles at the Apple Store.
Intermediate:	In Beijing, mourners and admirers came to lay flowers and light candles at the Apple Store.
Elementary:	In Beijing, people went to the Apple Store with flowers and candles.
Advanced:	There are signs that, in Britain and America, streaming may soon generate more revenue for the music industry than downloads from online stores such as Apples iTunes.
Intermediate:	In Britain and America, streaming may soon generate more revenue for the music industry than downloads from online stores such as Apples iTunes.
Elementary:	In Britain and America, streaming may soon make more money for the music industry than downloads from online stores such as Apples iTunes.

Table 6.1: Example sentences at different reading levels. First group has sentences with different sentence lengths. Second group has two of the sentences with same sentence length.

first block of Table 6.1. This sentence is complex compared to elementary sentence. But it is simpler than the advanced sentence in the same group. So, we can say that a sentence is simple or complex in the context of other sentence. Also, a simpler sentence need not be shorter than the complex sentence. For example, consider the sentences in the second block. Sentences in intermediate and elementary reading levels have same sentence length. But the elementary sentence has simpler words like ‘make’, ‘money’ where as the intermediate sentence has relatively complex words like ‘generate’, ’revenue’. So complexity can be both at the lexical level like simple vs. complex words as well as at the syntactic level like the presence/absence of co-ordination, relative clauses etc. These examples clearly show that treating a sentence pair as a single entity is better than treating each sentence in the pair as independent sentences.

Noting that not all SIMPLEWIKI sentences are simpler than every other sentence in WIKI, Vajjala (2015) re-framed the problem as a ranking problem according to which given a pair of parallel SIMPLEWIKI and WIKI sentences, the former must be ranked better than the latter in terms of readability. Inspired by Vajjala (2015), we also treat each pair together, and model relative readability assessment as a pairwise classification problem. Let a, b be a pair of parallel sentences. Let $\mathbf{a}$, $\mathbf{b}$ represent their corresponding feature vectors. We define our classifier Φ as

$$\begin{aligned} \Phi(\mathbf{a}-\mathbf{b}) = \quad & 1 && \text{if } a \in \text{SIMPLE } \& \, b \in \text{COMPLEX} \\ = \, & -1 && \text{if } b \in \text{SIMPLE } \& \, a \in \text{COMPLEX} \end{aligned}$$

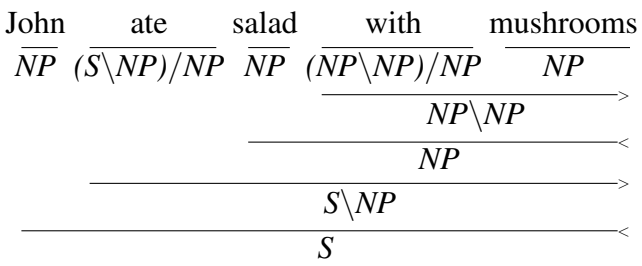
The motivation for our modelling is that relative features (difference) are more useful than absolute features, e.g., intuitively shorter sentences are simple to read, but length can only be defined in comparison with another sentence. A value of 20 for sentence length doesn't give much information. A 20 word sentence can be a SIMPLEWIKI sentence in one pair and a WIKI sentence in another pair. But, if we take the difference in the length of the sentences in the pair, then it correlates much better with the class label. A positive value correlates with *COMPLEX* class and a negative value with *SIMPLE* class, which shows that WIKI sentences are longer than SIMPLEWIKI sentences in most of the cases.

6.3 Incremental CCG Parse Features

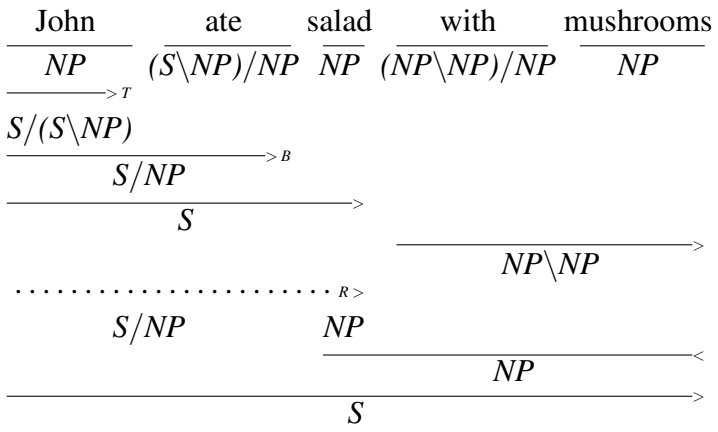
In this section, we provide necessary background, and then present the features explored in our experiments.

6.3.1 CCG vs. PST

Figure 6.1(b) displays an incremental CCG derivation. Here, the syntactic type (category) *(S\NP)/NP* on *ate* indicates that it is a transitive verb looking for a *NP* (object) on the right-hand side and a *NP* (subject) on the left-hand side. In Figure 6.1(b), the category of *with (NP\NP)/NP* combines with the category of *mushrooms NP* on its right-hand side using the combinatory rule of *forward application* (indicated by $>$), to form the category *NP\NP* representing the phrase *with mushrooms*. This phrase in



(a) Normal form Derivation



(b) Incremental Derivation

Figure 6.1: Normal form and incremental CCG derivations for an example sentence.

turn combines with other contextual categories using CCG combinators to form new categories representing larger phrases.

In contrast to phrase structure trees (PST), CCG derivation trees encode a richer notion of syntactic type and constituency. For example, in a phrase structure tree, the category (constituency tag) of *ate* would be *VBD* irrespective of whether it is transitive or intransitive, whereas the CCG category distinguishes these types. As the linguistic complexity increases, the complexity of the CCG category may increase, e.g., the relative pronoun has the category *(NP\NP)/(S\NP)* in relative clause constructions. In addition, CCG derivation trees have combinators annotated at each level which indicate the way in which the category is derived, e.g., in Figure 6.1(b) the category S/NP of *John ate* is formed by first *type-raising* (indicated by >T) *John* and then applying *forward composition* (indicated by >B) with *ate*. CCG combinators can throw light into the linguistic complexity of the construction, e.g., *crossed composition* is an indicator of long-range dependency. Phrase structure trees do not have this additional information encoded on their nodes.

6.3.2 Incremental CCG

In Chapter 4, we introduced a transition-based incremental CCG parser for English.[2] The main difference between this incremental version and standard non-incremental CCG parsers such as Zhang and Clark (2011a) is that as soon as the grammar allows two types to combine, they are greedily combined. For example, Figure 6.1(a) presents a normal-form derivation used by non-incremental parsers for an example sentence 'John ate salad with mushrooms'. Whereas, Figure 6.1(b) presents the incremental derivation for the same sentence. In the normal-form derivation, all the words in the sentence are first shifted to the stack. Then a preposition phrase 'with mushrooms' if formed which is then attached to the object 'salad'. Then the transitive verb 'ate' resolves its object 'salad' first and then the subject 'John'.

In the case of incremental derivation, first *John* is pushed on the stack but is immediately reduced when its head *ate* appears on the stack (i.e., *John*'s category combines with *ate*'s category to form a new category), and similarly when *salad* is seen, it is reduced with *ate*. When *with* appears it waits to be reduced until its head *mushrooms* appears on the stack, and later *mushrooms* is reduced with *salad* via *ate* using a spe-

[2]This parser is not word by word (strictly) incremental but is incremental with respect to CCG derivational constituents following the Strict Competence Hypothesis (Steedman, 2000). See Chapter 4 for complete details.

Figure 6.2: Incremental Derivation for a relatively complex sentence.

Figure 6.3: Incremental Derivation for a relatively simple sentence.

cial *revealing* operation (indicated by R>) followed by a sequence of operations. The *revealing* operation is performed when a category has greedily consumed a head in advance of a subsequently encountered post-modifier to regenerate the head. In the non-incremental version, *salad* is not reduced with *ate* until *with mushrooms* is reduced with it.

Consider the following sentences A and B where B is a simpler version of A.

A	Mourners and admirers came to lay flowers and light candles at the Apple Store.
B	People went to the Apple Store with flowers and candles.

Figures 6.2 and 6.3 present the incremental derivations for both these sentences. Consider the CCG category for 'to' in both the sentences. In A, the category of 'to' is *(S[dcl]\NP)/(S[to]\NP)* which is much complex compared to the category of 'to' in B which is PP/NP. Both the derivations have one right reveal action (indicated by $R>$). In A, the depth of this action is two since it is VP coordination. Whereas, in B the depth is only one. Section 6.3.3.2 provides the details about the depth feature. This kind of information can be useful in predicting the complexity of a sentence.

6.3.3 Features

As discussed above, as the complexity of a sentence increases, the complexity of CCG categories, combinators and the number of revealing operations increase in the incremental analysis. We exploit this information to assess readability of a sentence. For each sentence, we build a feature vector using the features defined below extracted from its incremental CCG derivation.

6.3.3.1 Sentence Level Features

These features include sentence length, height of the CCG derivation, and the final number of fragments. Sentence length is the number of words in the sentence. Since the CCG derivation is a binary tree, height of the derivation tree is the number of edges from the root node to the deepest leaf node. A CCG derivation can have multiple constituents if none of the combinators allow the constituents to combine. This happens mainly in ungrammatical sentences. These uncombined constituents are called fragments. If the parser gives a fully connected derivation tree then there

is only one constituent and hence no fragments. If the parser couldn't find a complete spanning analysis, then the derivation will have multiple fragments. A complex sentence is expected to have more number of fragments compared to a simpler sentence, which could be a useful feature for predicting sentence complexity. Below are the list of sentence level features used for our experiments.

- Sentence Length
- Height of the Derivation Tree
- Number of fragments

6.3.3.2 CCG Rule Counts

These features include the number of applications, forward applications, backward applications, compositions, forward compositions, backward compositions, left punctuations, right punctuations, coordinations, type-raisings, type-changing, left revealing, right revealing operations used in the CCG derivation. Each combinator is treated as a different feature dimension with its count as the feature value. For the revealing operations, we also add additional features which indicate the depth of the revealing which is analogous to surprisal (Hale, 2001)[3]. *Depth* here is defined based on the attachment of the node in the revealing action. *Depth=1* indicates that the attachment is with the bottom most node. Similarly, *depth=2* indicates that the attachment is with the second bottom node and so on. Below is the list of all the 16 features in this category.

- Application (3): Forward, Backward, All
- Composition (3): Forward, Backward, All
- Punctuation (3): Left, Right, All
- Others (4): Coordination, Type-raising, Type-changing Rules
- Reveal (3): Left Reveal, Right Reveal, Depth of Reveal

[3]Please see Chapter 4 for additional information on the *depth* of revealing operations.

6.3.3.3 CCG Categories

We define the complexity of a CCG category as the number of basic syntactic types used in the category, e.g., the complexity of *(S[pss]\NP)/(S[to]\NP)* is 4 since it has one *S[pss]*, one *S[to]*, and two *NP*s. Note that CCG type *S[pss]* indicates a *sentence* but of the subtype *passive*. We use average complexity of all the CCG categories used in the derivation as a real valued feature. In addition, we define integer-valued features representing the frequency of specific subtypes (we have 20 subtypes each defined as a different dimension) and the frequency of the top 8 syntactic types (each as a different dimension). All 29 features in this category are listed below.

- Average number of arguments
- Sentence subtypes (16): declarative (dcl), passive (pss) etc.
- Noun subtypes (4): Expletive *it* (exp), *there* (thr), non-bare (nb), numbers (num).
- Other (8): # NPs, # VPs, # PPs, # Ss, # noun modifiers, # verbal modifiers, # transitive verbs, # intransitive verbs.

6.4 Experimental Setup

In this section, we describe the setup for our experiments. We present the details about the dataset and the classifier used. We also provide information about our baseline model which uses non-incremental phrase structure features from the Stanford parser.

6.4.1 Evaluation Data

As evaluation data, we use WIKI and SIMPLEWIKI parallel sentence pairs collected by Hwang et al. (2015), a newer and larger version compared to Zhu et al. (2010)'s collection. We only use the pairs from the section GOOD consisting of 150K pairs. We further removed pairs containing identical sentences which resulted in 117K clean pairs. We randomly divided the data into training (60%), development (20%) and test (20%) splits.

6.4.2 Implementation details

As our classifier (see Section 6.2) we use SVM with Sequential Minimal Optimization in Weka toolkit (Hall et al., 2009) following its popularity in readability litera-

Model	Beam	Look-ahead	Accuracy
NON-INCREMENTAL PST	-	Yes	71.68
GREEDY NON-INCREMENTAL CCG	No	Yes	72.03
GREEDY INCREMENTAL CCG	No	Yes	**72.12**
INCREMENTAL CCG	Yes	No	71.97

Table 6.2: Impact of different syntactic features.

ture (Feng, 2010; Hancke et al., 2012; Vajjala and Meurers, 2014). We use polynomial kernel and default settings for hyperparameters. This is similar to Vajjala and Meurers (2014), but defers from Vajjala (2015) which uses SVMRank [4] (Joachims, 2006). We use the parsers described in Chapter 4 for extracting CCG derivations. For supertagging we use EasyCCG supertagger[5] (Lewis and Steedman, 2014b) since it gave better results than the C&C supertagger[6] (Clark, 2002).

6.4.3 Baseline: NON-INCREMENTAL PST

Following Vajjala (2015), we use features extracted from Phrase Structure Trees (PST) produced by the Stanford parser[7] (Klein and Manning, 2003), a non-incremental parser. We use the exact code used by Vajjala (2015) to extract these features which include part-of-speech tags, constituency features like the number of noun phrases, verb phrases and preposition phrases, and the average size of the constituent trees. Vajjala (2015) used a total of 57 features.[8]

6.5 Results

First we analyze the impact of CCG features from different CCG parsers. We experiment with the greedy non-incremental CCG parser (we call this GREEDY NON-INCREMENTAL CCG) and two versions of the incremental CCG parser described in

[4] http://www.cs.cornell.edu/people/tj/svm_light/svm_rank.html
[5] http://homepages.inf.ed.ac.uk/s1049478/easyccg.html
[6] http://svn.ask.it.usyd.edu.au/trac/candc/
[7] http://nlp.stanford.edu/software/lex-parser.html
[8] Details of the features can be found in Vajjala (2015).

Model	Beam	Look-ahead	Accuracy
Vajjala and Meurers (2014)	-	Yes	62.63
Vajjala (2015)	-	Yes	74.58
NON-INCREMENTAL PST++	-	Yes	78.68
GREEDY NON-INCREMENTAL CCG++	No	Yes	78.77
GREEDY INCREMENTAL CCG++	No	Yes	**78.87**
INCREMENTAL CCG++	Yes	No	78.77

Table 6.3: Performance of models with both syntactic and psycholinguistic features.

Chapter 4: greedy parser which uses look-ahead features (we call this GREEDY INCREMENTAL CCG) and a beam-search parser which doesn't use a look-ahead (we call this INCREMENTAL CCG). Table 6.2 presents the results of predicting relative readability on the test data.[9] GREEDY INCREMENTAL CCG achieves the best accuracy of 72.12%, a significant[10] improvement of 0.44 points over NON-INCREMENTAL PST (71.68%) indicating that incremental CCG features are empirically more useful than non-incremental phrase structure features. INCREMENTAL CCG gave an accuracy of 71.97% which is slightly lower than GREEDY INCREMENTAL CCG but better than the baseline NON-INCREMENTAL PST. Parsing accuracy of INCREMENTAL CCG is slightly lower than GREEDY INCREMENTAL CCG which could be the reason for this.

We also evaluate if this result holds for incremental vs. non-incremental CCG parse features. Note that in the non-incremental version, revealing features are absent. This version achieves an accuracy of 72.02%, around 0.1% lower than the winner GREEDY INCREMENTAL CCG, yet higher than NON-INCREMENTAL PST showing that CCG derivation trees offer richer syntactic information than phrase structure trees.

Apart from the syntactic features, Vajjala (2015) have also used psycholinguistic features such as age of acquisition of words, word imagery ratings, word familiarity ratings, and ambiguity of a word, collected from the psycholinguistic repositories

[9] All feature engineering is done on the development data.

[10] Numbers in bold indicate significant results, significance measured using McNemar's test.

Celex (Baayen et al., 1995), MRC (Wilson, 1988), AoA (Kuperman et al., 2012) and WordNet (Fellbaum, 1998). These features are found to be highly predictive for assessing readability. We enhance our syntactic models NON-INCREMENTAL PST, GREEDY NON-INCREMENTAL CCG, GREEDY INCREMENTAL CCG and INCREMENTAL CCG by adding these psycholinguistic features to build NON-INCREMENTAL PST++, GREEDY NON-INCREMENTAL CCG++, GREEDY INCREMENTAL CCG++, INCREMENTAL CCG++ respectively. Table 6.3 presents the final results along with the previous work of Vajjala and Meurers (2014) and Vajjala (2015). We ran their code on our dataset and the results are similar to their results on Zhu et al. (2010)'s dataset. Psycholinguistic features gave a boost of around 6.75 points on the syntactic models. Additionally the performance gap between our models decrease (from 0.44 to 0.19) showing some of the psycholinguistic features also model a subset of the syntactic features. GREEDY INCREMENTAL CCG++ achieves the best accuracy of 78.77% outperforming the previous best system of Vajjala (2015) by a wide margin.

We see similar trend in the results before and after adding psycholinguistic features. In general, CCG features gave better results than phrase structure features. Among different CCG parsers, GREEDY INCREMENTAL CCG parser performed consistently better than GREEDY NON-INCREMENTAL CCG parser. Between both the versions of the incremental parsers, GREEDY INCREMENTAL CCG parser, which is a greedy parser using a look-ahead gave better than the INCREMENTAL CCG parser, which uses a beam and no look-ahead. Also, GREEDY INCREMENTAL CCG parser is around ten times faster than the INCREMENTAL CCG parser making it more appealing for the practical real-time applications.

6.5.1 Speed

In addition to accuracy, parsing speed is important in real-time applications. Table 6.4 presents the speed comparison for NON-INCREMENTAL PST and GREEDY INCREMENTAL CCG parsers. The Stanford parser (NON-INCREMENTAL PST) took 204 minutes to parse the test data with a speed of 3.8 sentences per second. The GREEDY INCREMENTAL CCG parser took 16 minutes with an average speed of 47.5 sentences per second, a 12X improvement over the Stanford parser. These numbers include POS tagging time for the Stanford parser, and POS tagging and supertagging time for the GREEDY INCREMENTAL CCG parser. This shows that our GREEDY INCREMENTAL CCG parser is both efficient (12 times faster) and accurate (0.44 points

Model	Speed (Sentences/Second)
NON-INCREMENTAL PST	3.8
GREEDY INCREMENTAL CCG	**47.5**

Table 6.4: Speed comparison.

better) in comparison to the previously used Stanford parser for this task.

6.6 Conclusion

Our empirical evaluation on assessing relative sentence complexity suggests that syntactic features extracted from an incremental CCG parser are more useful than from a non-incremental phrase structure parser. This result aligns with psycholinguistic findings that human sentence processor is incremental. Our incremental model enhanced with psycholinguistic features achieves the best reported results on predicting relative sentence readability. We explored a greedy incremental parser which uses a look-ahead and a beam-search parser without a look-ahead of which the former was both accurate and efficient for this task. Similarly, for other tasks as well we can explore different levels of incrementality and choose the one suitable for the task.

We experimented with Simple Wikipedia and Wikipedia data from Hwang et al. (2015). We can explore the usefulness of our system on other datasets like OneStopEnglish (OSE) corpus (Vajjala, 2015) or the dataset from Xu et al. (2015b).

Chapter 7

Transition-based CCG Parsing using Neural Network Models

In this chapter, we present a neural network based transition-based CCG parser, one of the first neural-network based parsers for CCG [1]. We also study the impact of neural network based tagging models, and greedy versus beam-search parsing, by using a structured neural network model. We experiment with both English and Hindi CCGbanks. For English, our greedy parser obtains a labelled F-score of 83.27%, the best reported result for greedy CCG parsing in the literature (an improvement of 2.5% over a perceptron based greedy parser) and is more than three times faster. For Hindi, our greedy parser achieves a labelled F-score of 74.14% which is an improvement of 3% over the greedy perceptron parser. In the beam search case, the structured neural network model, though not the state-of-the-art, consistently gave better results than the basic neural network model. Part of this work is published in Ambati et al. (2016b).

7.1 Introduction

Shift-reduce parsing is interesting for practical real-world applications like parsing the web, since parsing can be achieved in linear time. Although greedy parsers are fast, accuracies of these parsers are typically much lower than graph-based parsers. Conversely, beam-search parsers achieve accuracies comparable to graph-based parsers (Zhang and Nivre, 2011) but are much slower than their greedy counterparts. Recently, Chen and Manning (2014) have showed that fast and accurate

[1] At the same time, and independent of this thesis, Xu et al. (2016) developed a neural network based CCG parser.

parsing can be achieved using neural network based parsers. Improving their work, Weiss et al. (2015) presented deep neural network and structured neural network models which gave state-of-the-art results for English dependency parsing.

Extending this recent work on neural network based dependency parsers, we present a neural network based shift-reduce CCG parser, one of the first neural network based parsers for CCG. We first adapt Chen and Manning (2014)'s shift-reduce dependency parser for CCG parsing. We then develop a structured neural network model based on Weiss et al. (2015), in order to explore the impact of a beam-search on the parser. We also analyze the impact of neural network taggers (for both POS-tagging and CCG supertagging) as compared to maximum entropy taggers.

For English, over a perceptron based greedy shift-reduce parser our neural network parser achieves an improvement of 2.3% and 2.4% in unlabelled and labelled F-scores respectively. We observe further improvements when we use neural network taggers instead of the standard maximum entropy taggers. We experimented with both non-incremental and the incremental versions of the parsing algorithm described in chapter 4 and the non-incremental algorithm gave better than the incremental algorithm. We achieve final results of 89.78% and 83.27% unlabelled and labelled F-scores respectively on the standard CCGbank test data with a non-incremental greedy parser using neural network taggers. These are the best reported results for greedy CCG parsing in the literature. Additionally, our neural network parser is more than three times faster than its perceptron counterpart making it appealing for practical real-world applications. Even for Hindi, in the greedy settings, our neural network parser gave 3% improvements over the perceptron parser.

When we use a beam, structured neural network model gave better results than the the basic neural network model. To the best of our knowledge, ours is the first neural network based parser for CCG and also the first work on exploring neural network taggers for shift-reduce CCG parsing. This parser is available for public usage at `https://bitbucket.org/bharatambati/tranccg`.

The rest of the chapter is arranged as follows. Section 7.2 gives a brief introduction to related work in the areas of word embeddings and neural network parsing. Section 7.3 presents our neural network based parser for English. Details of the experiments and analysis of the results on English CCGbank are provided in section 7.4. Hindi parser, experiments and results are presented in sections 7.5 and 7.6. We conclude with possible future directions in section 7.7.

7.2 Related Work

Related work in CCG parsing is already provided in Section 4.2.1. Here we present related work in the areas of word embeddings and neural network parsing.

7.2.1 Word Embeddings

There has been increasing interest in using continuous vectors in the form of word embeddings rather than discrete words as features. Word embeddings not only generalize well but also help in minimizing the feature engineering required for the task. Word embeddings are successfully applied to a wide variety of NLP tasks such as POS-tagging, supertagging, chunking, named-entity recognition, semantic role labeling, phrase structure parsing and dependency parsing (Turian et al., 2010; Collobert et al., 2011; Collobert, 2011; Socher et al., 2013; Chen and Manning, 2014; Lewis and Steedman, 2014b; Xu et al., 2015a).

Our work is closely related to Lewis and Steedman (2014b) and Xu et al. (2015a). Lewis and Steedman (2014b) first developed a neural network based CCG supertagger using feed-forward architecture. They explored different publicly available word embeddings and achieved best results with Turian embeddings (Turian et al., 2010). Then they used this supertagger in the state-of-the-art graph based CCG parser (C&C) and showed improvements on both in-domain and out-of-domain test sets. Extending Lewis and Steedman (2014b), Xu et al. (2015a) developed a recurrent neural network based supertagger. They showed improvements over the feed-forward based supertagger for both supertagging and parsing using the C&C parser. We also explore the impact of neural network based CCG supertagger, especially Lewis and Steedman (2014b)'s EasyCCG tagger, for shift-reduce CCG parsers.

7.2.2 Neural Network Parsers

Neural Network parsers are attracting interest due to both speed and accuracy. There has been some work on neural networks for constituent based parsing (Collobert, 2011; Socher et al., 2013; Watanabe and Sumita, 2015). Chen and Manning (2014) developed a neural network architecture for dependency parsing. This parser was fast and accurate, parsing around 1000 sentences per second and achieving an unlabelled attachment score of 92.0% on the standard Penn Treebank test data for English. Chen and Manning (2014)'s parser used a feed forward neural network. Several improve-

ments were made to this architecture in terms of using Long Short-Term Memory (LSTM) networks (Dyer et al., 2015), deep neural networks (Weiss et al., 2015) and structured neural networks (Weiss et al., 2015; Zhou et al., 2015; Alberti et al., 2015).

Weiss et al. (2015) first trained a neural network similar to Chen and Manning (2014). Then they trained a structured perceptron model with a beam which takes as input the pre-trained neural networks hidden layer(s) and the output layer. This system is called a structured neural network. They also used a tri-training technique for semi-supervised training using raw text. In tri-training, two parsers are run on the raw text and the parse trees of the sentences for which both parsers gave same analysis are used as training data. Weiss et al. (2015) also used two hidden layers with 2048 hidden units in each layer compared to Chen and Manning (2014)'s single hidden layer with 200 hidden units. With their structured neural network model, Weiss et al. (2015) obtained the state-of-the-art results for English dependency parsing. Alberti et al. (2015) extended Weiss et al. (2015)'s parser by introducing set-valued features for capturing morphological information and part-of-speech confusion sets. They also explored the impact of joint POS tagging and dependency parsing. In addition to English, they experimented with CoNLL 2009 Shared Task languages (Hajič et al., 2009) and showed the usefulness of their approach.

We explore Chen and Manning (2014) style feed-forward neural network and structured neural network of Weiss et al. (2015) for our work. Unlike Weiss et al. (2015) who use a deep neural network with two hidden layers for pre-training[2], we use a feed-forward neural network with one hidden layer for pre-training. We provide the details of our neural network parsers in the following sections.

7.3 Our Neural Network Parser (NNPar): English

The architecture of our neural network based shift-reduce CCG parser is similar to that of Chen and Manning (2014). We present the details of the network and the model settings in this section. We also discuss our structured neural network model.

7.3.1 Layers

Figure 7.1 shows the architecture of our neural network parser. There are three layers in our parser: the input, hidden and output layers. We first extract the discrete features

[2]Following the neural network literature, we define any network with more than one hidden layer as deep neural network.

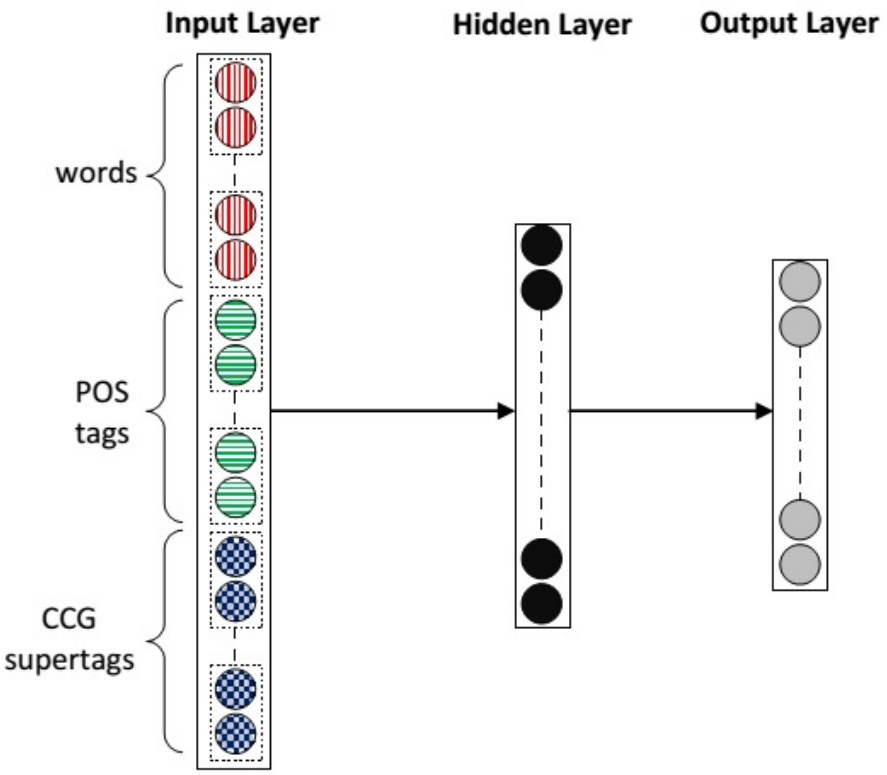


Figure 7.1: Our Neural Network Architecture (adapted from Chen and Manning (2014)).

from the parser configuration, such as words, POS-tags and CCG supertags. For each of these discrete features we obtain a continuous vector representation in the form of their corresponding embeddings and use them in the input layer.

The input layer is mapped to a hidden layer using the following function:

$$h = (W_h * E + b_h)^3$$

where E is the input embedding vector, W_h is the weight matrix and b_h is the bias vector. Following Chen and Manning (2014), we use a cube activation function. They used *tanh*, *sigmoid*, *identity*, and *cube* activation functions and showed that the cube activation function achieves better results over other functions since it can capture the interaction of three elements, such as tri-gram features, in a better way.

The output layer is a standard softmax layer and uses the following function:

$$Y = softmax(W_y * h)$$

where h is the hidden layer vector and W_y is the weight matrix. Y is the final vector which represents probabilities for each class in the multi-class classification.

7.3.2 Feature and Model Settings

We extract features from a) top four nodes in the stack (S0, S1, S2, S3), b) next four nodes in the input (Q0, Q1, Q2, Q3) and c) left and right children of the top two nodes in the stack (S0L, S0R, S1L, S1R). We obtain words (w) and POS-tags (p) of all 12 nodes. In the case of CCG supertags, in addition to the CCG categories (c) of the nodes in the stack (top four nodes, left and right children of top two nodes), we also obtain the lexical head categories (S0Hc, S1Hc) for the top two nodes. For example, in Figure 4.6, after step 6, *S\NP:loves* is the top node in the stack (S0). CCG category and lexical head category of this node (S0c and S0Hc) are *S\NP* and *(S\NP)/NP* respectively. So, in total we have 34 features: 12 word, 12 POS-tag and 10 CCG supertag features, presented in Table 7.1. We use a special token 'NULL' if a feature is not present in the parser configuration.

Types	*Features*
words	S0w, S1w, S2w, S3w, Q0w, Q1w, Q2w, Q3w, S0Lw, S0Rw, S1Lw, S1Rw,
POS-tags	S0p, S1p, S2p, S3p, Q0p, Q1p, Q2p, Q3p, S0Lp, S0Rp, S1Lp, S1Rp,
CCG supertags	S0c, S1c, S2c, S3c, S0Lc, S0Rc, S1Lc, S1Rc, S0Hc, S1Hc

Table 7.1: The deature templates of our parser.

For each of these 34 features we obtain their corresponding embeddings. Lewis and Steedman (2014b) explored different publicly available word embeddings (Mnih and Hinton, 2009; Turian et al., 2010; Collobert et al., 2011; Mikolov, 2012; Mikolov et al., 2013a) for CCG supertagging and showed that Turian embeddings of dimensionality 50 (Turian-50) gave better results. We explored embeddings from Collobert et al.

(2011) and Turian et al. (2010) and we also got better results with Turian-50 embeddings. So, we use Turian-50 embeddings for words in our parser. For the words which are not in the word embeddings dictionary, embeddings of '-UNKNOWN-' token are used as a backoff. Note that this '-UNKNOWN-' token is different from the 'NULL' token used for non-existent features. For POS-tags and CCG supertags, the parameters are randomly initialized within (-0.01, 0.01).

Our input layer is a 34 (feature templates) $\times$ 50 (embedding size) dimensional vector. We use 200 hidden units in the hidden layer. For the output layer we compute softmax probabilities only for the actions which are possible in a particular parser configuration instead of all the 2296 actions. We use the training settings of Chen and Manning (2014) for our parser. The training objective is to minimize the cross-entropy loss with an l_2-regularization and the training error derivatives are backpropagated during training. For optimization we use AdaGrad (Duchi et al., 2011). 10^{-8} and 0.01 are the values for regularization parameter and Adagrad initial learning rate respectively. To make the parser faster, matrix multiplications are pre-computed for the most frequent 10,000 words. Parameters that give the best labelled F-score on the development data are used for testing data.

7.3.3 Structured Neural Network

Chen and Manning (2014)'s parser is a greedy parser, and it is not straightforward to add a beam during training into their parser. As a way of introducing a beam, Weiss et al. (2015) presented structured perceptron training for the neural network parser, known as a structured neural network. Following Weiss et al. (2015), we first pre-train the feed-forward neural network model described in the previous section. For the final layer, we train a structured perceptron using beam search decoding which takes the neural network's hidden and output layers as the input. In addition to using a softmax for the output layer, we also applied this structured neural network approach for our experiments using a beam. Unlike Weiss et al. (2015)'s neural network architecture, which consists of two hidden layers with 2048 hidden units each, we use the Chen and Manning (2014) style architecture described in the previous sections.

7.3.4 Comparison to Chen and Manning (2014)

Our neural network parser is adapted from Chen and Manning (2014) and differs from their work in a number of respects. We use CCG supertags in the input layer since

ours is a CCG parser. Chen and Manning (2014) use dependency label set as theirs is a dependency parser. For word embeddings, we use Turian embeddings (Turian et al., 2010) whereas they use embeddings from Collobert et al. (2011). We have slightly smaller set of 34 feature templates compared to their 48 templates. Our parser has 2296 actions while Chen and Manning (2014) has much smaller number of actions (35 for CoNLL and 91 for Stanford dependencies). For CCG, the shift action performs category disambiguation by assigning a CCG category, in addition to shifting the node. Also, there are many more CCG categories (500) compared to dependency labels (50) which resulted in more operations.

7.4 Experiments and Results: English

We first compare our neural network parser (NNPar)[3] with a perceptron based parser in the greedy setting. Then we study the impact of incremental algorithm and neural network taggers for transition-based CCG parsing. Next we analyze the impact of a beam using neural network (NNPar) and structured neural network (Structured NNPar) models.

The perceptron based parser is a re-implementation of Zhang and Clark (2011a)'s parser (Z&C*). A global linear model trained with the averaged perceptron (Collins, 2002) is used for this parser and an early-update (Collins and Roark, 2004) strategy is used during training. In the greedy setting (beam=1), when the predicted action differs from the gold action, decoding stops and the weights are updated accordingly. When a beam is used (beam=16), the weights are updated when the gold parse configuration falls out of the beam. For Z&C*, the feature set of Zhang and Clark (2011a), which comprises of 64 feature templates is used. For NNPar, the 34 feature templates described in section 7.3.2 are used.

7.4.1 Data and Settings

We use the standard CCGbank training (sections $02-21$), development (section 00) and testing (section 23) splits for our experiments. All the experiments are performed using automatic POS-tags and CCG supertags. We compare the performance using two types of taggers: maximum entropy and neural network based taggers (NNT). The C&C taggers [4] (Clark and Curran, 2004b) are used for maximum entropy taggers.

[3] We used Chen and Manning (2014)'s package for implementing our NNPar

[4] http://svn.ask.it.usyd.edu.au/trac/candc/wiki

For neural network taggers, SENNA tagger[5] (version 3.0) (Collobert et al., 2011) is used for POS-tagging and EasyCCG tagger[6] (Lewis and Steedman, 2014a) is used for supertagging. Both of these taggers use a feed-forward neural network architecture with a single hidden layer similar to our NNPar architecture.

In the case of POS-tags, we consider the first best tag given by the POS tagger. For CCG supertags, we use a multitagger which gives n-best supertags for a word. Following Zhang and Clark (2011a) and Xu et al. (2014), only during training, the gold CCG lexical category is added to the list of supertags for a word if it is not present in the list assigned by the multitagger.

7.4.2 Parsing Model

In this section, we compare the performance of the perceptron based parser (Z&C*) and neural network based parser (NNPar). We explore the greedy setting (beam=1) in this section. Both parsers use C&C taggers.

Table 7.2 presents the unlabelled F-score (UF), labelled F-score (LF) and lexical category accuracy (Cat Acc.) for the Z&C* and NNPar on the CCGbank development data. NNPar outperformed Z&C* on all the metrics. There is significant improvement of 2.14% in UF and 2.4% in LF. This observation is in line with the Chen and Manning (2014)'s dependency parsing results. They obtained improvements of about 2% in the unlabelled and labelled attachment scores for their neural network based parser over their perceptron based baseline parser. Since our NNPar is based on Chen and Manning (2014)'s parser, our results show that their neural network architecture is robust enough to be successfully applied across different grammatical formalisms.

Model	*UF*	*LF*	*Cat Acc.*
Z&C*	87.24	80.25	91.09
NNPar	**89.38**	**82.65**	**91.72**

Table 7.2: The performance of the Perceptron (Z&C*) and Neural Network (NNPar) parsers.

[5] http://ronan.collobert.com/senna/

[6] http://homepages.inf.ed.ac.uk/s1049478/easyccg.html

7.4.3 Parsing Algorithm

In Chapter 4 we presented the non-incremental algorithm (NonInc) of Zhang and Clark (2011a) which is a standard shift-reduce CCG parsing algorithm and the revealing based incremental algorithm (RevInc) which uses special revealing actions. We train our NNPar using both parsing algorithms in a greedy setting. NonInc has ∼2300 actions whereas RevInc has ∼3000 actions. This made the parser very slow and computationally expensive. So, we pruned out the sentences with less frequent actions. Actions which occurred less than two times in the entire training data are considered as less frequent actions which pruned around 400 actions. As a result, 4% of the training sentences are removed and the remaining 96% of the data is used for training. For NonInc we used all the sentences without any pruning.

The results with both the algorithms are presented in Table 7.3. The first block (top two rows) in the table presents the results using NonInc algorithm which are the same as the results from Table 7.2. The second block (last two rows) presents the results with RevInc algorithm. Similar to NonInc, the neural network model gave better results than the perceptron model. There is an improvement of around 0.3% in both UF and LF which is lower than the improvements observed for NonInc. Pruning the actions based on the threshold for RevInc could be one of the reasons for this. Overall NonInc with the neural network model gave the best results. So, for the rest of the experiments we use NonInc parsing algorithm.

Parsing Algorithm	*Model*	*UF*	*LF*	*Cat Acc.*
NonInc	Z&C*	87.24	80.25	91.09
NonInc	NNPar	**89.38**	**82.65**	**91.72**
RevInc	Z&C*	88.69	80.75	90.87
RevInc	NNPar	89.08	81.07	90.86

Table 7.3: Performance of NonInc and RevInc parsing algorithms using Perceptron and Neural Network models.

7.4.4 Taggers

Maximum entropy based taggers of C&C use discrete features, while neural network based taggers like EasyCCG tagger use continuous vector features and hence generalize well (Lewis and Steedman, 2014a). Also, the C&C supertagger uses POS-tag features. As a result, errors made by the POS-tagger significantly affect the performance of the supertagger. EasyCCG supertagger avoids this problem as it doesn't use POS-tag features. Lewis and Steedman (2014a) showed that the performance of a state-of-the-art graph-based CCG parser like C&C parser can be significantly improved by replacing the C&C supertagger with EasyCCG supertagger. They obtained an improvement of 0.8% in labelled F-score with EasyCCG supertagger. Xu et al. (2015a) extended this work. They developed a recurrent neural network based supertagger and showed even better improvements for C&C parser. In this section, we observe the impact of neural network based taggers for shift-reduce CCG parsing. We use both a POS tagger and supertagger for our experiments, both neural network based.

Table 7.4 shows the impact of neural network taggers for NNPar in greedy settings. The neural network based taggers (NNT) improve over the C&C taggers by 0.7%. Our result is in line with Lewis and Steedman (2014a) and Xu et al. (2015a) and shows that neural network taggers improve the performance of shift-reduce CCG parsers as well. We obtained final unlabelled and labelled F-scores of 90.09% and 83.33% respectively on the development data. To the best of our knowledge these are the best reported results for greedy shift-reduce CCG parsing.

Taggers	*UF*	*LF*	*Cat Acc.*
C&C	89.38	82.65	91.72
NNT	**90.09**	**83.33**	**92.03**

Table 7.4: Impact of Neural Network based taggers on NNPar.

7.4.5 Beam Search

In section 7.4.2, we showed that a neural network based parser (NNPar) outperforms a perceptron based parser (Z&C*) in greedy settings. We observed a further boost in the performance of the parser using neural network based taggers (NNT) in section 7.4.4. Now, we analyze the impact of these neural network based parsing and tagging

Model	*Beam*	*Taggers*	*UF*	*LF*	*Cat Acc.*
Z&C*	1	C&C	87.24	80.25	91.09
	1	NNT	87.00	79.78	90.52
	16	C&C	91.17	84.34	92.42
	16	NNT	**92.10**	**85.75**	**93.05**
NNPar	1	C&C	89.38	82.65	91.72
	1	NNT	90.09	83.33	92.03
	16	C&C	90.78	83.76	91.98
	16	NNT	91.46	84.55	92.35
Structured NNPar	16	NNT	**92.19**	**85.69**	**93.02**
Zhang and Clark (2011a)	16	C&C	-	85.00	92.77
Xu et al. (2014)	128	C&C	-	**85.18**	**92.75**

Table 7.5: Impact of the beam on Perceptron and Neural Network based parsers. Number is bold are best systems in the respective blocks.

models for the beam search parsers. For Z&C* and Structured NNPar, following Zhang and Clark (2011a), we use a beam of size 16 both during training and testing. But for NNPar, we use a beam of 16 only during testing. During the training phase, the neural network model of NNPar is trained locally.

Table 7.5 presents the results of Z&C*, NNPar and structure NNPar parsers using a beam size of 16. The first block of the table (top 3 rows) presents the results for Z&C*. The second block of the table (middle 4 rows) presents the results for our neural network parsers. In this second block, the first three rows are the results with our NNPar and the last row presents the results with our Structured NNPar. The last block (last 2 rows) presents the published results of Zhang and Clark (2011a) and Xu et al. (2014).

For Z&C*, using a beam improved the unlabelled and labelled F-scores by around 4% over the greedy parser with C&C taggers. Using neural network taggers (NNT) gave further improvements of 0.9% and 1.4% in UF and LF respectively. We obtain

UF and LF of 92.10% and 85.75% respectively on the development data.

For NNPar, using a beam improved the unlabelled and labelled F-scores by 1.4% and 1.1% respectively over the greedy parser with C&C taggers. Using neural network taggers (NNT) gave further improvements of 0.7% and 0.8% in UF and LF respectively. We obtain UF and LF of 91.46% and 84.55% respectively on the development data. Similar to Z&C*, using a beam gave better results than the greedy parser and using neural network taggers gave further boost over the C&C taggers for NNPar as well.

Our NNPar uses a beam only during training. Zhang and Nivre (2012) showed that for dependency parsing using a beam during training as well as testing gives much better results than using it only during testing. To overcome this problem, we experimented with our Structured NNPar model described in section 7.3.3. We achieve F-score of 92.19% in UF and 85.69% in LF with our Structured NNPar which is an improvement of 1.1% in LF over NNPar. This shows that in the case of neural network parsing, structured neural network model performs better than the basic neural network model. This is similar to the result of Weiss et al. (2015) for dependency parsing. Notice that both the structured perceptron (Z&C*) and structured neural network (Structured NNPar) give almost similar results when they both use NNT taggers. To the best of our knowledge these are the best reported results for shift-reduce CCG parsing on the development data.

7.4.6 Final Test Results

Table 7.6 presents the results for the final test data. The first block of the table (top 4 rows) presents the results in the greedy settings. The second block of the table (middle 4 rows) presents the results with a beam. The last block (last 2 rows) presents the published results of Zhang and Clark (2011a) and Xu et al. (2014).

With the greedy setting, NNPar outperformed Z&C* in all the cases. Also NNT gave slight improvements over C&C for NNPar. Final best results of 89.78% in UF and 83.27% in LF are obtained with NNPar which are the best reported result for greedy shift-reduce CCG parsing.

In the case of the beam search parsers, we achieved final best scores of 91.76% in UF and 85.59% in LF for Z&C* using NNT. We observed improvements of 0.5% in both UF and LF by using NNT over C&C. In the case of neural network models, we got an accuracy of 91.95% in UF and 85.57% in LF using our Structured NNPar, an improvement of 1.1% in LF over the NNPar. We get the best category accuracy with

Parser	*Beam*	*Taggers*	*UF*	*LF*	*Cat Acc.*
Z&C*	1	C&C	87.28	80.78	91.44
Z&C*	1	NN	86.44	79.78	90.67
NNPar	1	C&C	89.61	83.18	92.04
NNPar	1	NNT	**89.78**	**83.27**	**91.89**
Z&C*	16	C&C	91.28	85.00	92.79
Z&C*	16	NNT	91.76	**85.59**	92.84
NNPar	16	NNT	91.14	84.44	92.22
Structured NNPar	16	NNT	**91.95**	85.57	**92.86**
Zhang and Clark (2011a)	16	C&C	-	85.48	92.77
Xu et al. (2014)	128	C&C	-	**86.00**	92.75

Table 7.6: Results on the CCGbank test data.

our Structured NNPar but LF is slightly lower than Xu et al. (2014). Note however that we use a much smaller beam size of 16 (similar to Z&C) compared to theirs (128). Results are presented with beam=16 to enable direct comparison with Zhang and Clark (2011a), since our parsing algorithm is similar to theirs. Increasing the beam size improved the accuracy but significantly reduced the parsing speed. Testing with beam=128 gave 0.2% improvement in accuracy (still 0.2 points lower than Xu et al. (2014)) but slowed the parser by ten times.

7.4.7 Label-wise Impact

In this section, we analyze the impact of different models on the top 10 most frequent CCG supertags. Table 7.7 presents the F-scores of different greedy parsers for these supertags. The first two columns show the results for Z&C* and NNPar using C&C taggers. NNPar gives better results than Z&C* for all the CCG supertags. For adjuncts like $(NP\backslash NP)/NP$, $((S\backslash NP)\backslash(S\backslash NP))/NP$, there are improvements of

1-2%. Much higher improvements of 3-4% are observed for verbal arguments like *((S[dcl]\NP)/NP*. NNT (third column) gave better results than C&C for NNPar in most of the cases (7/10). For adjuncts, both C&C and NNT shared the top spots. But for verbal arguments, NNT gave significant improvements of 1-3%.

Table 7.8 presents the F-scores of the top 10 most frequent CCG categories for beam search parsers. The first two columns show the results for Z&C* using C&C and NNT taggers. Similar to greedy parsers, NNT gave better results than C&C for most (8/10) of the cases. Especially for verbal arguments, there are improvements of 2-3% in the F-score. Last two columns present the result for NNPar and Structured NNPar using NNT taggers. Structured NNPar gave better results than NNPar for all the categories. But the results are slightly lower than Z&C* using NNT tagger since the overall labelled F-score for Structured NNPar is slightly lower than Z&C.

Category	*Z&C*-C&C*	*NNPar-C&C*	*NNPar-NNT*
*N/**N***	94.35	94.85	**95.80**
*NP[nb]/**N***	95.88	96.11	**96.82**
*(NP\NP)/**NP***	81.84	83.49	**83.91**
*(NP**NP**)/NP*	81.81	**83.10**	82.56
*((S\NP)\(S\NP))/**NP***	67.04	**69.26**	69.23
*((S\NP)\(**S\NP**))/NP*	66.06	68.85	**69.85**
*((S[dcl]\NP)/**NP***	77.32	81.62	**82.72**
*PP/**NP***	65.16	67.25	**69.24**
*((S[dcl]**NP**)/NP*	74.13	77.12	**80.33**
*(S\NP)\(**S\NP**)*	83.40	**85.00**	83.33

Table 7.7: Label-wise F-score of different systems on the top 10 most frequent CCG categories in greedy settings. Argument slots in the relation are in bold.

Category	*Z&C*-C&C*	*Z&C*-NNT*	*NNPar-NNT*	*Structured NNPar-NNT*
$N/\mathbf{N}$	95.00	**96.24**	95.77	96.01
$NP[nb]/\mathbf{N}$	96.84	**97.53**	96.90	97.46
$(NP\backslash NP)/\mathbf{NP}$	82.54	**84.97**	84.65	84.82
$(NP\backslash \mathbf{NP})/NP$	83.10	**83.78**	83.25	83.39
$((S\backslash NP)\backslash (S\backslash NP))/\mathbf{NP}$	**71.79**	71.56	69.39	70.62
$((S\backslash NP)\backslash (\mathbf{S}\backslash \mathbf{NP}))/NP$	69.67	**70.81**	69.74	70.01
$((S[dcl]\backslash NP)/\mathbf{NP}$	84.17	86.00	84.75	**87.07**
$PP/\mathbf{NP}$	71.10	**71.82**	70.03	71.31
$((S[dcl]\backslash \mathbf{NP})/NP$	82.28	**85.73**	83.31	85.48
$(S\backslash NP)\backslash (\mathbf{S}\backslash \mathbf{NP})$	**85.79**	85.37	84.25	85.75

Table 7.8: Label-wise F-score of different systems on the top 10 most frequent CCG categories for beam search parsers.

7.4.8 Speed

Beam-search parsers are more accurate than greedy parsers but are very slow. With neural network models we can build parsers which give a nice trade-off between speed and accuracy. Table 7.9 present the speed comparison for both Z&C* and our NNPar in greedy settings. NNPar is much faster, parsing 350 sentences per second compared to Z&C* which parses 110 sentences per second. Parsers with beam=16 parse around 10 sentences per second and parsers with beam=128 parse around 1 sentence per second. These numbers don't include tagging time.

Model	*Sentences/Second*
Z&C*	110
NNPar	350

Table 7.9: Speed comparison of perceptron and neural network based greedy parsers.

7.5 Our Neural Network Parser (NNPar): Hindi

The architecture of our neural network based shift-reduce CCG parser for Hindi is similar to that of English described in section 7.3. Our neural network parser for Hindi has two major differences from the English parser. The first concerns the input layer. For English, the input layer has embeddings of words, pos-tags and CCG supertags. For Hindi, in addition to these, we also provide embeddings of lemma, coarse pos-tag, morphological features available in the LEMMA, CPOSTAG, FEATS columns in the CoNLL format. As we have seen previously in Chapter 5 and as shown in the literature (Ambati et al., 2010a,b), morphological features play a crucial role in parsing Hindi. So, we provided embeddings for all the columns in the CoNLL format. The second difference is that we created word embeddings using Word2Vec[7] rather than using publicly available embeddings like Turian-50 for English. We provide the details below.

Figure 7.2 presents the neural network architecture for Hindi CCG parsing. Similar to English, it has three layers: input embedding layer, hidden layer and output layer.

[7] https://code.google.com/archive/p/word2vec/

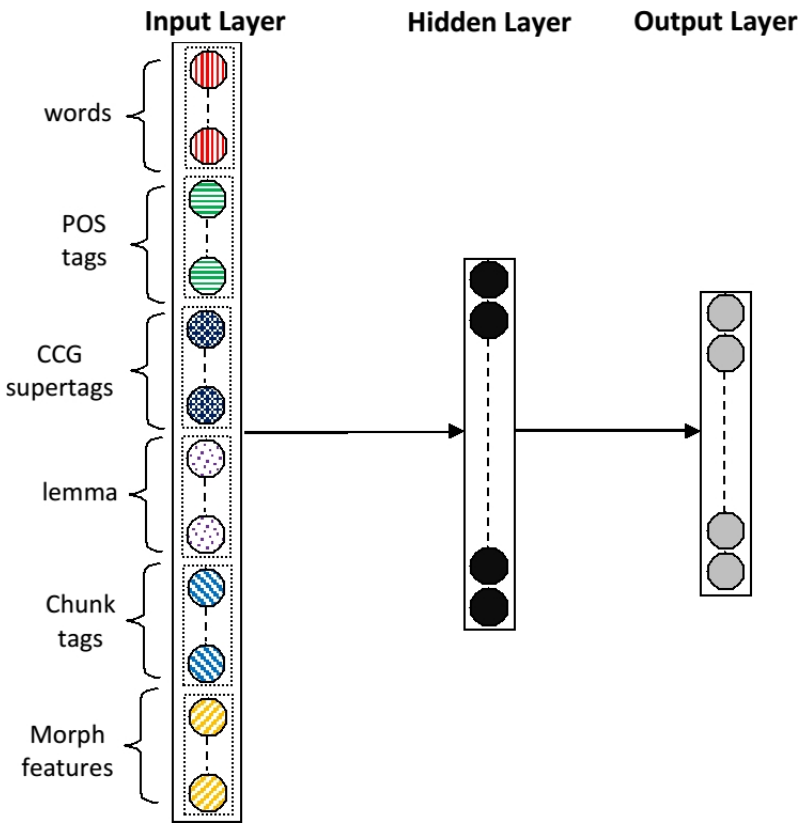


Figure 7.2: Our Neural Network Architecture for Hindi parsing.

Apart from the input layer, all other parameters like activation function, number of hidden units, learning rate etc. are same as the English parser.

7.5.1 Features

We extract features from a) the top four nodes in the stack (S0, S1, S2, S3), b) the next four nodes in the input (Q0, Q1, Q2, Q3) and c) the left and right children of the top two nodes in the stack (S0L, S0R, S1L, S1R). We obtain the words (w) and POS-tags (p) of all these 12 nodes. As for the CCG supertags, in addition to the CCG categories (c) of the nodes in the stack (the top four nodes, the left and right children of the top two nodes), we also obtain the lexical head categories (S0Hc, S1Hc) for the top two nodes.

All the above features are common to the feature set used for English. In addition to these features we extract lemma (l) and coarse POS-tags (t) for the top two nodes in the stack (S0, S1) and the next two nodes in the input (Q0, Q1). We also obtain morphological features (m) for the top two nodes in the stack (S0, S1). Each node has three morphological features in the form of suffix, case marker and head/non-head

node of the chunk. So, in total we have 48 features: 12 word, 12 POS-tag, 10 CCG supertag, 4 lemma, 4 coarse POS-tag and 6 morphological features. Table 7.1 gives the list of word, POS-tag and CCG category features for the English parser which is common to Hindi parser. The list of all the additional features used for Hindi CCG parsing is presented in Table 7.10. We use a special token 'NULL' if a feature is not present in the parser configuration.

Types	*Features*
lemma	S0l, S1l, Q0l, Q1l
coarse POS-tags	S0t, S1t, Q0t, Q1t
Morphological Features	S0m, S1m

Table 7.10: Additional feature templates for Hindi CCG parser.

7.5.2 Word Embeddings

For each of these 48 features we obtain their corresponding embeddings. Since there are no publicly available word embeddings for Hindi, we created word embeddings using Word2Vec (Mikolov et al., 2013a,b). For words not in the word embeddings dictionary, the embedding of an '-UNKNOWN-' token is used as a backoff. For other features (lemma, POS-tags, coarse POS-tags, morphological features, CCG supertags) the parameters are randomly initialized within (-0.01, 0.01). We use 50 dimensional word embeddings for our experiments. As a result our input layer is a 48 (feature templates) $\times$ 50 (embedding size) dimensional vector.

7.6 Experiments and Results: Hindi

Similar to the English experiments, we first compare our neural network parser (NNPar) with a perceptron based parser in the greedy setting. Then we analyze the impact of a beam using neural network (NNPar) and structured neural network (Structured NNPar) models. Since we don't have neural network taggers for Hindi we don't do any experiments studying their impact.

7.6.1 Data and Settings

We use the Hindi CCGbank training, development and testing splits for our experiments. We experiment with both gold and automatic POS-tag and chunk features. In both cases, CCG supertags are automatically assigned using word, POS-tag, chunk tag and other features. The details of the Hindi supertagger are presented in 5.4.2. Our experiments in Chapter 5 showed that because of sparsity issues and smaller amounts of training data, coarse-grained lexicon gave better parsing results than the fine-grained lexicon. So, we experiment only with coarse-grained lexicon in this section.

Similar to the English experiments, in the case of POS-tags, we consider the first best tag given by the POS tagger. For CCG supertags, instead of a supertagger which provides a single best supertag, we use a multitagger which gives n-best supertags for a word. Following Zhang and Clark (2011a) and Xu et al. (2014), only during training, the gold CCG lexical category is added to the list of supertags for a word if it is not present in the list assigned by the multitagger. All feature tuning is done on the development data and the settings which gave the best results on the development data are directly used for the testing data.

7.6.2 Parsing Model

In this section, we compare the performance of perceptron based parser (Z&C*) and neural network based parser (NNPar). We explore the greedy setting (beam=1) in this section. Both parsers use maximum entropy taggers.

Table 7.11 presents the unlabelled F-score (UF), labelled F-score (LF) and lexical category accuracy (Cat Acc.) for the Z&C* and NNPar on the Hindi CCGbank testing data. First block (first three columns) presents the results with gold features and the second block (last three columns) shows the results with automatic features. NNPar outperformed Z&C* on all the metrics in both settings. With automatic features we obtained the final best results of 83.57% in UF and 74.14% in LF with our NNPar. This is an improvement of 2.14% in UF and 2.97% in LF over Z&C*. This observation is inline with the Chen and Manning (2014)'s dependency parsing results and similar to the results we obtained for English.

Model	*Gold*			*Auto*		
	UF	*LF*	*Cat Acc.*	*UF*	*LF*	*Cat Acc.*
Z&C*	85.47	75.89	87.55	81.43	71.17	83.83
NNPar	**86.56**	**77.98**	**89.12**	**83.57**	**74.14**	**86.29**

Table 7.11: Impact of neural network model on greedy Hindi CCG parsing.

Model	*Beam*	*Gold*			*Auto*		
		UF	*LF*	*Cat Acc.*	*UF*	*LF*	*Cat Acc.*
Z&C*	1	85.47	75.89	87.55	81.43	71.17	83.83
NNPar	1	**86.56**	**77.98**	**89.12**	**83.57**	**74.14**	**86.29**
Z&C*	16	**89.12**	**81.74**	**91.16**	**85.60**	**77.32**	**87.64**
NNPar	16	87.74	79.23	89.62	84.75	75.52	86.95
Structured NNPar	16	88.10	79.94	90.05	85.10	75.86	86.98

Table 7.12: Impact of beam on Hindi CCG parsing.

7.6.3 Beam Search

Next, we analyze the impact of beam-search on various parsers. For Z&C* and Structured NNPar, we use a beam of size 16 both during training and testing; for NNPar, a beam (of 16) can be used only during testing. Table 7.12 presents the results using a beam size of 16. For comparison we also present the results of the greedy parsers. The top 2 rows of the table show the results of greedy parsers and the last 3 rows present the results of beam-search parsers.

Using a beam improved the performance of both the perceptron and neural network parsers. Improvements are much larger for Z&C* (6%) compared to NNPar (1-2%). Since NNPar uses a beam only during testing, there is only a slight improvement in the F-score. Using a structured neural network gave further improvements but the Z&C* gave the best results. We obtained a final LF of 77.32% with automatic features using Z&C*. Structured NNPar performs better than NNPar even for Hindi. But it didn't give F-scores comparable to Z&C* for Hindi. Unlike English, we are capturing much richer information in the form of word, lemma, POS-tag, coarse POS-tag and morphological features with just 200 hidden units. Maybe improving the neural network architecture and the number of hidden units similar to Weiss et al. (2015) and exploring better ways of providing morphological features like Alberti et al. (2015) might give better results for Hindi.

7.6.4 Label-wise Impact

Table 7.13 presents the results for the top 10 most frequent CCG categories for Hindi. The first two columns show the results for greedy parsers described in 7.6.2. The last three columns present the results for beam-search parsers described in 7.6.3. For all the categories NNPar gave better results than Z&C* in the greedy settings. There are improvements of around 3% for NP conjuncts or genitive markers ($(NP/NP)\backslash NP$) and Post-position markers for adjunct nouns ($(S_f/S_f)\backslash NP$). As for the beam, Z&C* gave better F-scores for most of the categories (8/10). There are minor differences in the performance between Z&C* and Structured NNPar for the top eight categories. But the difference is significant in the case of verbal arguments ($(S_f\backslash NP)\backslash NP$).

Category	Greedy		Beam		
	*Z&C**	*NNPar*	*Z&C**	*NNPar*	*Structured NNPar*
$NP/\mathbf{NP}$	79.01	**82.36**	83.67	82.76	**83.72**
$S_f\backslash\mathbf{S}_f$	84.04	**87.20**	91.07	90.48	**91.19**
$NP\backslash\mathbf{NP}$	81.10	**82.72**	**83.49**	82.94	82.44
$(S_f/S_f)\backslash\mathbf{NP}$	77.73	**78.54**	**81.91**	78.77	80.74
$(S_f/\mathbf{S}_f)\backslash NP$	73.08	**76.43**	**79.87**	77.65	79.14
$(NP/NP)\backslash\mathbf{NP}$	77.93	**80.10**	**82.17**	80.47	81.00
$(NP/\mathbf{NP})\backslash NP$	73.26	**76.68**	**77.55**	77.05	77.51
$S_f/\mathbf{S}_f$	68.30	**72.45**	**73.53**	72.30	73.50
$(S_f\backslash NP)\backslash\mathbf{NP}$	61.87	**64.28**	**67.62**	66.06	65.89
$(S_f\backslash\mathbf{NP})\backslash NP$	58.17	**63.61**	**68.55**	67.04	65.89

Table 7.13: Label-wise F-score for top 10 most frequent CCG categories for Hindi.

7.6.5 Speed

Similar to English, our NNPar is more efficient in terms of parsing speed in the greedy settings. Table 7.14 presents the speed comparison for both Z&C* and our NNPar. NNPar is much faster, parsing 485 sentences per second compared to Z&C* which parses 210 sentences per second. Note that similar to English, these numbers only include parsing time and do not include tagging time.

7.7 Conclusion

We presented the first neural network based shift-reduce parsers for CCG, a greedy and a beam-search parser. We explored neural network based tagging models as well as parsing models, and a structured neural network model. For English, on the standard CCGbank test data, we achieved a labelled F-score of 85.57% with our structured neu-

Model	*Sentences/Second*
Z&C*	210
NNPar	485

Table 7.14: Speed comparison of perceptron based and neural network based parsers.

ral network parser which gave comparable accuracies to structured perceptron parser (85.59%). Our greedy parser gets UF and LF of 89.78% and 83.27% respectively, the best reported results for a greedy CCG parser, and is more than three times faster than its perceptron counterpart. We also found that neural network taggers gave better results than maximum entropy based C&C taggers for our experiments.

We observed similar results with Hindi parsing as well. In greedy settings, the neural network parser is both more efficient (more than two times faster) and more accurate (3% better) than the perceptron parser. When we use a beam, the structured neural network gave better results than the basic neural network model. However, the structured perceptron gave the best results compared to the structured neural network model for Hindi.

In general, in greedy settings, the neural network parser performed better than the perceptron parser. With the use of a beam, the structured neural network parser performed better than the basic neural network parser. Also, neural network taggers gave better results than the maximum entropy taggers. In the case of a beam, the structured neural network model gave comparable accuracies to the structured perceptron model for English and slighlty lower results for Hindi. In the future we plan to explore more complex models like deep neural networks with more than one hidden layer (Weiss et al., 2015), and recurrent neural networks (Dyer et al., 2015) for CCG parsing.

Chapter 8

Conclusion

We presented an incremental algorithm for parsing Combinatory Categorial Grammar (CCG) for two diverse languages: English and Hindi in this thesis. English is a fixed word order and morphologically simple language, whereas, Hindi is a free word order and morphologically rich language. We showed that CCG is incremental enough for developing a practically useful incremental parser. Also, ours is the first broad coverage incremental parser for a verb final language, Hindi. Our algorithm builds a dependency graph in parallel to the CCG derivation which is used for revealing the unbuilt structure. Though we used dependencies for meaning representation and CCG for parsing, our revealing technique can be applied to other meaning representations like lambda expressions and for non-CCG parsing like phrase structure parsing. This thesis was divided into three parts each dealing with a major module of the statistical parser: data (Chapters 2, 3), parsing algorithm (Chapters 4, 5, 6) and learning algorithm (Chapter 7).

In Chapter 2, we presented an approach for automatically creating a CCGbank from a dependency treebank for Hindi. We created two types of lexicon: fine-grained which keeps morphological information in noun and verb categories and coarse-grained which doesn't. We provided a detailed analysis of various long-range dependencies like coordinate and relative constructions, and showed how to handle them in CCG. Our approach for converting dependency treebanks to CCGbanks has already been successfully applied to Telugu, another Indian language (Kumari and Rao, 2015). Our approach is generic enough to extract CCG lexicons and/or CCGbanks for the many other languages for which dependency treebanks are available, including the languages of the CoNLL dependency parsing shared tasks (Buchholz and Marsi, 2006; Nivre et al., 2007a) and universal dependency treebanks (McDonald et al., 2013).

In Chapter 3, We showed that informative CCG categories improve the performance of dependency parsers like Malt and MST. CCG categories improved the overall accuracy of both parsers by around 0.3-0.5% in all experiments. Our experiments showed that adding CCG categories as features to these dependency parsers helped in recovering long distance relations for Malt and verbal arguments for MST. This result is particularly interesting in the case of Malt, a greedy parser, as we get improvements in performance without compromising speed and hence can be applicable to web scale processing. Our results apply both to English, a fixed word order and morphologically simple language, and to Hindi, a free word order and morphologically rich language, indicating that CCG categories from a supertagger are an easy and robust way of introducing lexicalized subcategorization information into dependency parsers. Though we worked on English and Hindi, our approach can be applied for other languages such as Turkish, German etc. for which both dependency and CCG resources are available. Kumari and Rao (2015) already used our technique to improve Telugu dependency parser.

We presented a novel algorithm for incremental transition-based CCG parsing for English and Hindi in chapters 4 and 5. We introduced two new actions into the shift-reduce paradigm which reveal the unbuilt structure. We presented two versions of incremental parsers. The first one is a greedy parser which uses a look-ahead. And the second one is a beam-search parser which doesn't use a look-ahead. The first one gives a nice trade-off between accuracy and speed and hence is useful for practical real-time applications. Whereas the second one is more useful for psycholinguistic studies. Our incremental algorithm models transitions rather than incremental derivations, and hence we don't need an incremental CCGbank. Our approach can therefore be adapted to languages with dependency treebanks, since CCG lexical categories can be easily extracted from dependency treebanks (Cakici, 2005). We presented the first and basic version of the incremental algorithm for parsing Hindi, a verb final language. We can improve the algorithm to handle morphological richness and free word order nature of Hindi by using a lexicon similar to Baldridge (2002) and by introducing new actions for handling coordination.

In Chapter 6 we showed how these incremental parsers can be useful for a practical application like assessing relative sentence complexity. Given a pair of sentences from wikipedia and simple wikipedia, we built a classifier which predicts if one sentence is simpler or more complex than the other. We showed that features from a CCG parser in general and incremental CCG parser in particular are more useful than a chart-based

phrase structure parser both in terms of speed (12 times faster) and accuracy (0.44% better). In this work we only studied the impact of incremental parsers for practical applications. In future we can analyze the usefulness of our incremental parser for psycholinguistic studies.

Finally in Chapter 7, we presented the first neural network based transition-based parser for CCG. We explored neural network based tagging models as well as parsing models, and a structured neural network model. For English, on the standard CCGbank test data, we achieved a labelled F-score of 85.57% with our structured neural network parser which gave comparable accuracies to a structured perceptron parser (85.59%). Our greedy parser gave unlabelled and labelled F-scores of 89.78% and 83.27% respectively, the best reported results for a greedy CCG parser, and is more than three times faster. We observed similar improvements for Hindi CCG parsing as well. In general in greedy settings, the neural network parser performed better than the perceptron based parser. With the use of a beam, structured neural network parser performed better than the basic neural network parser. Also, neural network taggers gave better results than maximum entropy taggers. We plan to explore more complex models like recurrent neural networks (Dyer et al., 2015) and deep neural networks (Weiss et al., 2015) in future.

Related Publications

1. **Bharat Ram Ambati**, Tejaswini Deoskar, and Mark Steedman. (2016). Hindi CCGbank: CCG Treebank from the Hindi Dependency Treebank. In *Language Resources and Evaluation* (Under Review).

2. **Bharat Ram Ambati**, Tejaswini Deoskar, and Mark Steedman. (2016). Shift-Reduce CCG Parsing using Neural Network Models. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

3. **Bharat Ram Ambati**, Siva Reddy, and Mark Steedman. (2016). Assessing Relative Sentence Complexity using an Incremental CCG Parser. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

4. **Bharat Ram Ambati**, Tejaswini Deoskar, Mark Johnson and Mark Steedman. (2015). An Incremental Algorithm for Transition-based CCG Parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 53-63, Denver, Colorado.

5. **Bharat Ram Ambati**, Tejaswini Deoskar and Mark Steedman. (2014). Improving Dependency Parsers using Combinatory Categorial Grammar. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers*, pages 159-163, Gothenburg, Sweden.

6. **Bharat Ram Ambati**, Tejaswini Deoskar and Mark Steedman. (2013). Using CCG categories to improve Hindi dependency parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 604-609, Sofia, Bulgaria.

Appendix A

Hindi Tagset and CCGbank Format

A.1 Hindi POS, chunk and dependency tagset

The following is the list of pos tags, chunk tags and dependency labels used in Hindi treebank. For complete description, see the pos, chunk[1] and dependency guidelines[2]

Sl No.	Category	Tag name
1.1	Noun	NN
1.2	Locative Noun	NST
2.	Proper Noun	NNP
3.1	Pronoun	PRP
3.2	Demonstrative	DEM
4	Verb-finite	VM
5	Verb Aux	VAUX
6	Adjective	JJ
7	Adverb	RB

[1] http://ltrc.iiit.ac.in/MachineTrans/publications/technicalReports/tr031/posguidelines.pdf

[2] http://ltrc.iiit.ac.in/MachineTrans/research/tb/DS-guidelines/DS-guidelines-ver2-28-05-09.pdf

8	Post position	PSP
9	Particles	RP
10	Conjuncts	CC
11	Question Words	WQ
12.1	Quantifiers	QF
12.2	Cardinal	QC
12.3	Ordinal	QO
12.4	Classifier	CL
13	Intensifier	INTF
14	Interjection	INJ
15	Negation	NEG
16	Quotative	UT
17	Symbols	SYM
18	Compounds	*C
19	Reduplicative	RDP
20	Echo	ECH
21	Unknown	UNK

Table A.1: Hindi POS Tagset

Sl. No	Chunk Type	Tag Name
1	Noun Chunk	NP
2.1	Finite Verb Chunk	VGF

2.2	Non-finite Verb Chunk	VGNF
2.3	Infinitival Verb Chunk	VGINF
2.4	Verb Chunk (Gerund)	VGNN
3	Adjectival Chunk	JJP
4	Adverb Chunk	RBP
5	Chunk for Negatives	NEGP
6	Conjuncts	CCP
7	Chunk Fragments	FRAGP
8	Miscellaneous	BLK

Table A.2: Hindi Chunk Tagset

Hindi depenency label	**English Equivalent**	**Description**
k1 (kartha)	SUBJ	Subject/Agent
k1s (samanadhikarana)	SCOM	Noun complements of kartha
k2 (karma)	OBJ	Object/Patient
k3 (karana)	INST	Instrument
k4 (sampradaana)	RCPT	Recipient
k5 (apaadaana)	SRC	Source
k7t (kaalaadhikarana)	TIME	Time Expression
k7p (deshadhikarana)	PLACE	Place Expression
r6 (shashthi)	GEN	Possessive/Genitive marker

nmod_relc	RELC	Relative Clause
vmod	VMOD	Verbal Modifier
nmod	NMOD	Noun Modifier
nmod__adj	AMOD	Adjectival modifier of a noun
lwg__psp	CASE	Case marker
lwg__aux	AUX	Auxiliary verb or Tense, Aspect and Modality marker for verb
pof	POF	Part-OF units such as conjunct verbs
rs	NELB	Noun Elaboration
r6-k1	CSUB	SUBJ of conjunct verb
r6-k2	COBJ	OBJ of conjunct verb

Table A.3: Hindi dependency labels and their English equivalents.

A.2 Hindi CCGbank: Machine-readable Format

CCG derivation for the first sentence in the Hindi dependency treebank guidelines using fine-grained lexicon is given below. We follow the format of Hockenmaier and Steedman (2007) for representing the binary CCG derivation trees with the bracketed notation.

(< T S_f 1 2> (< T *NP[ne]* 0 2> (< L *NP* NNP NNP raam *NP*>) (< L *NP[ne]*$\backslash$*NP* PSP PSP ne *NP[ne]*$\backslash$*NP*>)) (< T $S_f\backslash$*NP[ne]* 1 2> (< T *NP[ko]* 0 2> (< L *NP* NNP NNP mohan *NP*>) (< L *NP[ko]*$\backslash$*NP* PSP PSP ko *NP[ko]*$\backslash$*NP*>)) (< T $(S_f\backslash$*NP[ne])$\backslash$*NP[ko]* 1 2> (< T *NP[0]* 1 2> (< L *NP*/*NP* JJ JJ niilii *NP*/*NP*>) (< L *NP[0]* NN NN kitaab *NP[0]*>)) (< L $((S_f\backslash$*NP[ne])$\backslash$*NP[ko])$\backslash$*NP[0]* VM VM dii $((S_f\backslash$*NP[ne])$\backslash$*NP[ko])$\backslash$*NP[0]*>))))

There are two types of nodes in the derivation trees: Leaf nodes and Non-leaf

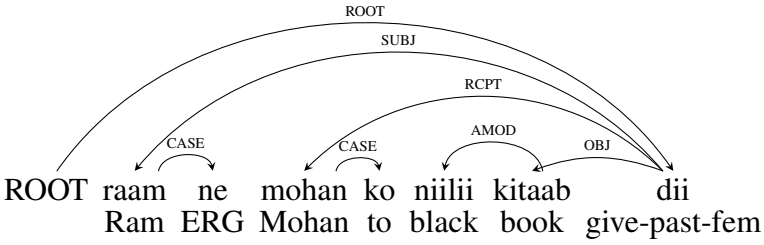
nodes. Leaf nodes have six fields.

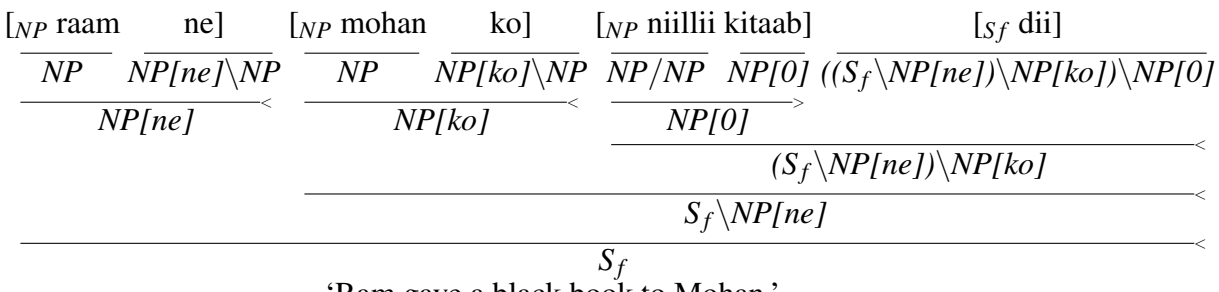
```
<L NP[ne] NNP NNP raam NP[ne]>
<L CCGCat mod-POS-tag orig-POS-tag word CCGCat2>
```

`L` represents that it is a leaf node. `CCGCat` is the CCG category of the node. Unlike English, POS tag is not modified during the conversion of dependency trees to CCG derivations. So, in Hindi CCGbank, `mod-POS-tag` and `orig-POS-tag` both represent the POS tag of the word. Lexical item is represented using `word` field. In English CCGbank, `CCGCat2` slot is used to represent predicate-argument structure of the CCG category. In Hindi CCGbank, we just use the lexical CCG category to fill this slot.

Non-leaf nodes have four fields. `T` represents that the node is a non-leaf node. `CCGCat` is the CCG category of the node. `head` takes two values: 0 if the left node is the head and 1 if the right node is the head. Since the CCG derivation trees are binary trees, `children` field will have 1 or 2 based on whether there are one or two children. Example non-leaf node is given below.

```
<T NP[ne] 0 2
<T CCGCat head children
```



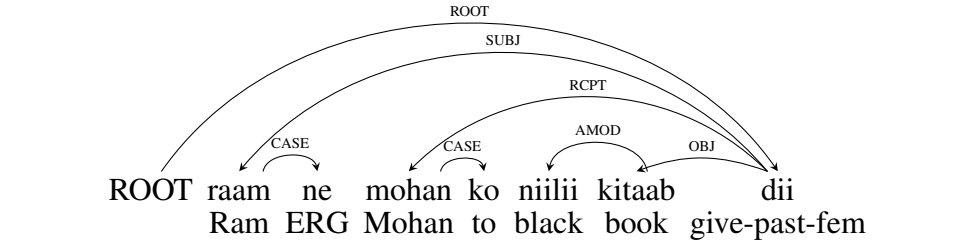


'Ram gave a black book to Mohan.'

Figure A.1: Example dependency tree and CCG derivation (Fine-grained).

CCG derivation tree with coarse-grained lexicon is provided below in machine readable format along with the dependency tree and derivation.

(< T S_f 1 2> (< T *NP* 0 2> (< L *NP* NNP NNP raam *NP*>) (< L *NP**NP* PSP PSP ne *NP**NP*>)) (< T S_f*NP* 1 2> (< T *NP* 0 2> (< L *NP* NNP NNP mohan *NP*>) (< L *NP**NP* PSP PSP ko *NP**NP*>)) (< T *(*S_f*NP)**NP* 1 2> (< T *NP* 1 2> (< L *NP*/*NP* JJ JJ niilii *NP*/*NP*>) (< L *NP* NN NN kitaab *NP*>)) (< L *((*S_f*NP)**NP)**NP* VM VM dii *((*S_f*NP)**NP)**NP*>))))



$[_{NP}$ raam ne] $[_{NP}$ mohan ko] $[_{NP}$ niillii kitaab] $[_{Sf}$ dii]

$\frac{\frac{NP \quad NP\backslash NP}{NP}< \quad \frac{NP \quad NP\backslash NP}{NP}< \quad \frac{\frac{NP/NP \quad NP}{NP}> \quad ((S_f\backslash NP)\backslash NP)\backslash NP}{(S_f\backslash NP)\backslash NP}<}{S_f\backslash NP}<}{S_f}<$

'Ram gave a black book to Mohan.'

Figure A.2: Example dependency tree and CCG derivation (Coarse-grained).

Bibliography

Alberti, C., Weiss, D., Coppola, G., and Petrov, S. (2015). Improved Transition-Based Parsing and Tagging with Neural Networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1354–1359, Lisbon, Portugal. Association for Computational Linguistics.

Altmann, G. and Steedman, M. (1988). Interaction with context during human sentence processing. *Cognition*, 30(3):191–238.

Ambati, B. R. (2011). *Hindi Dependency Parsing and Treebank Validation*. Master's Thesis, International Institute of Information Technology - Hyderabad, India.

Ambati, B. R., Deoskar, T., and Steedman, M. (2013). Using CCG categories to improve Hindi dependency parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 604–609, Sofia, Bulgaria. Association for Computational Linguistics.

Ambati, B. R., Deoskar, T., and Steedman, M. (2014). Improving Dependency Parsers using Combinatory Categorial Grammar. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers*, pages 159–163, Gothenburg, Sweden. Association for Computational Linguistics.

Ambati, B. R., Deoskar, T., and Steedman, M. (2016a). Hindi CCGbank: CCG Treebank from the Hindi Dependency Treebank. In *Language Resources and Evaluation (Under Review)*.

Ambati, B. R., Deoskar, T., and Steedman, M. (2016b). Shift-Reduce CCG Parsing using Neural Network Models. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 447–453, San Diego, California. Association for Computational Linguistics.

Ambati, B. R., Husain, S., Jain, S., Sharma, D. M., and Sangal, R. (2010a). Two Methods to Incorporate 'Local Morphosyntactic' Features in Hindi Dependency Parsing. In *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, pages 22–30, Los Angeles, CA, USA. Association for Computational Linguistics.

Ambati, B. R., Husain, S., Nivre, J., and Sangal, R. (2010b). On the Role of Morphosyntactic Features in Hindi Dependency Parsing. In *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, pages 94–102, Los Angeles, CA, USA. Association for Computational Linguistics.

Ambati, B. R., Reddy, S., and Steedman, M. (2016c). Assessing Relative Sentence Complexity using an Incremental CCG Parser. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1051–1057, San Diego, California. Association for Computational Linguistics.

Auli, M. and Lopez, A. (2011). 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: Human Language Technologies*, pages 470–480, Portland, Oregon, USA. Association for Computational Linguistics.

Avinesh, P. and Gali, K. (2007). Part-Of-Speech Tagging and Chunking using Conditional Random Fields and Transformation Based Learning. In *IJCAI-07 Workshop on Shallow Parsing in South Asian Languages*, pages 21–24, Hyderabad, India.

Baayen, R. H., Piepenbrock, R., and Gulikers, L. (1995). *The CELEX Lexical Database (CD-ROM)*. Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA.

Baldridge, J. M. (2002). *Lexically Specified Derivational Control in Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh, UK.

Begum, R., Husain, S., Bai, L., and Sharma, D. M. (2008a). Developing Verb Frames for Hindi. In *Proceedings of LREC*.

Begum, R., Husain, S., Dhwaj, A., Sharma, D. M., Bai, L., and Sangal, R. (2008b). Dependency annotation scheme for Indian languages. In *Proceedings of The Third International Joint Conference on Natural Language Processing (IJCNLP)*, pages 721–726, Hyderabad, India.

Bharati, A., Chaitanya, V., and Sangal, R. (1995). Natural Language Processing: A Paninian Perspective. *Prentice-Hall of India*, pages 65–106.

Bharati, A., Husain, S., Ambati, B., Jain, S., Sharma, D. M., and Sangal, R. (2008). Two semantic features make all the difference in Parsing accuracy. In *Proceedings of the 6th International Conference On Natural Language Processing (ICON)*, pages 11–19, Pune, India. Macmillan publishers India Ltd.

Bharati, A., Husain, S., Misra, D., and Sangal, R. (2009a). Two stage constraint based hybrid approach to free word order language dependency parsing. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09)*, pages 77–80, Paris, France. Association for Computational Linguistics.

Bharati, A., Mannem, P., and Sharma, D. M. (2012). Hindi Parsing Shared Task. In *Proceedings of Coling Workshop on Machine Translation and Parsing in Indian Languages*, Kharagpur, India.

Bharati, A., Sangal, R., and Sharma, D. M. (2007). SSF: Shakti Standard Format Guide. In *Technical Report (TR-LTRC-33), LTRC, IIIT-Hyderabad*.

Bharati, A., Sangal, R., Sharma, D. M., and Bai, L. (2006). AnnCorra: Annotating Corpora Guidelines for POS and Chunk Annotation for Indian Languages. In *Technical Report (TR-LTRC-31), LTRC, IIIT-Hyderabad*.

Bharati, A., Sharma, D. M., Husain, S., Bai, L., Begum, R., and Sangal, R. (2009b). AnnCorra: TreeBanks for Indian Languages, Guidelines for Annotating Hindi TreeBank (version 2.0). http://ltrc.iiit.ac.in/MachineTrans/research/tb/DS-guidelines/DS-guidelines-ver2-28-05-09.pdf.

Bhat, R. A. and Sharma, D. M. (2012). Non-projective structures in Indian language treebanks. In *Proceedings of the 11th Workshop on Treebanks and Linguistic Theories (TLT11)*, pages 25–30.

Bhatt, R., Narasimhan, B., Palmer, M., Rambow, O., Sharma, D. M., and Xia, F. (2009). A multi-representational and multi-layered treebank for Hindi/Urdu. In *Proceedings of the Third Linguistic Annotation Workshop at 47th ACL and 4th IJCNLP*, pages 186–189, Suntec, Singapore. Association for Computational Linguistics.

Bos, J., Bosco, C., and Mazzei, A. (2009). Converting a Dependency Treebank to a Categorial Grammar Treebank for Italian. In *Proceedings of the Eighth International Workshop on Treebanks and Linguistic Theories (TLT8)*, pages 27–38, Milan, Italy.

Bos, J., Clark, S., Steedman, M., Curran, J. R., and Hockenmaier, J. (2004). Wide-Coverage Semantic Representations from a CCG Parser. In *Proceedings of Coling 2004*, pages 1240–1246, Geneva, Switzerland. COLING.

Brants, S., Dipper, S., Hansen, S., Lezius, W., and Smith, G. (2002). The TIGER Treebank. In *Proceedings of the First Workshop on Treebanks and Linguistic Theories (TLT 2002)*, Sozopol, Bulgaria.

Buchholz, S. and Marsi, E. (2006). CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 149–164, New York City, New York. Association for Computational Linguistics.

Cakici, R. (2005). Automatic Induction of a CCG Grammar for Turkish. In *Proceedings of the ACL Student Research Workshop*, pages 73–78, Ann Arbor, Michigan. Association for Computational Linguistics.

Çakıcı, R. (2009). *Wide Coverage Parsing for Turkish*. PhD thesis, University of Edinburgh, UK.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Chelba, C. and Jelinek, F. (2000). Structured language modeling. *Computer Speech & Language*, 14(4):283–332.

Chen, D. and Manning, C. (2014). A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar. Association for Computational Linguistics.

Chu, Y. and Liu, T. (1965). On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400.

Clark, S. (2002). Supertagging for Combinatory Categorial Grammar. In *Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6)*, pages 19–24.

Clark, S. and Curran, J. (2006). Partial Training for a Lexicalized-Grammar Parser. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 144–151, New York City, USA. Association for Computational Linguistics.

Clark, S. and Curran, J. R. (2004a). Parsing the WSJ Using CCG and Log-Linear Models. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 103–110, Barcelona, Spain.

Clark, S. and Curran, J. R. (2004b). The Importance of Supertagging for Wide-Coverage CCG Parsing. In *Proceedings of Coling 2004*, pages 282–288, Geneva, Switzerland. COLING.

Clark, S. and Curran, J. R. (2007). Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. *Computational Linguistics*, 33:493–552.

Clark, S., Hockenmaier, J., and Steedman, M. (2002). Building Deep Dependency Structures using a Wide-Coverage CCG Parser. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 327–334, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Collins, M. (1997). Three Generative, Lexicalised Models for Statistical Parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid, Spain. Association for Computational Linguistics.

Collins, M. (1999). *Head-driven Statistical Models for Natural Language Parsing*. PhD thesis, University of Pennsylvania.

Collins, M. (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*, pages 1–8. Association for Computational Linguistics.

Collins, M. and Roark, B. (2004). Incremental Parsing with the Perceptron Algorithm. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 111–118, Barcelona, Spain.

Collins, M. J. (1996). A New Statistical Parser Based on Bigram Lexical Dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 184–191, Santa Cruz, California, USA. Association for Computational Linguistics.

Collobert, R. (2011). Deep learning for efficient discriminative parsing. In *International Conference on Artificial Intelligence and Statistics*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12:2493–2537.

Coppola, G. and Steedman, M. (2013). The Effect of Higher-Order Dependency Features in Discriminative Phrase-Structure Parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 610–616, Sofia, Bulgaria. Association for Computational Linguistics.

Coster, W. and Kauchak, D. (2011). Simple English Wikipedia: A New Text Simplification Task. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 665–669, Portland, Oregon, USA. Association for Computational Linguistics.

Covington, M. A. (2001). A fundamental algorithm for dependency parsing. In *In Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

Curran, J. R. and Clark, S. (2003). Investigating GIS and smoothing for maximum entropy taggers. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics-Volume 1*, pages 91–98. Association for Computational Linguistics.

Dalrymple, M., Shieber, S. M., and Pereira, F. C. (1991). Ellipsis and higher-order unification. *Linguistics and philosophy*, 14(4):399–452.

Darroch, J. N. and Ratcliff, D. (1972). Generalized iterative scaling for log-linear models. *The annals of mathematical statistics*, pages 1470–1480.

de Marneffe, M.-C., MacCartney, B., and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, pages 449–454.

Demberg, V. (2012). Incremental derivations in CCG. In *Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+ 11)*, pages 198–206.

Demberg, V. and Keller, F. (2008). Data from eye-tracking corpora as evidence for theories of syntactic processing complexity. *Cognition*, 109(2):193–210.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-Based Dependency Parsing with Stack Long Short-Term Memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240.

Eisner, J. (1996a). Efficient Normal-Form Parsing for Combinatory Categorial Grammar. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 79–86, Santa Cruz, California, USA. Association for Computational Linguistics.

Eisner, J. M. (1996b). Three New Probabilistic Models for Dependency Parsing: An Exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345, Copenhagen, Denmark. Association for Computational Linguistics.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). LIBLINEAR: A Library for Large Linear Classification. *The Journal of Machine Learning Research*, 9:1871–1874.

Fellbaum, C. (1998). *WordNet*. Wiley Online Library.

Feng, L. (2010). *Automatic readability assessment*. PhD thesis, City University of New York.

Foster, J., Cetinoglu, O., Wagner, J., Le Roux, J., Nivre, J., Hogan, D., and van Genabith, J. (2011). From News to Comment: Resources and Benchmarks for Parsing the Language of Web 2.0. In *Proceedings of 5th International Joint Conference on Natural Language Processing*, pages 893–901, Chiang Mai, Thailand. Asian Federation of Natural Language Processing.

Foth, K. A., By, T., and Menzel, W. (2006). Guiding a Constraint Dependency Parser with Supertags. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 289–296, Sydney, Australia. Association for Computational Linguistics.

Fowler, T. A. D. and Penn, G. (2010). Accurate Context-Free Parsing with Combinatory Categorial Grammar. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 335–344, Uppsala, Sweden. Association for Computational Linguistics.

Gadde, P., Jindal, K., Husain, S., Sharma, D. M., and Sangal, R. (2010). Improving data driven dependency parsing using clausal information. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 657–660, Los Angeles, California. Association for Computational Linguistics.

Galley, M., Graehl, J., Knight, K., Marcu, D., DeNeefe, S., Wang, W., and Thayer, I. (2006). Scalable Inference and Training of Context-Rich Syntactic Translation Models. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 961–968, Sydney, Australia. Association for Computational Linguistics.

Goldberg, Y. and Nivre, J. (2012). A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India. The COLING 2012 Organizing Committee.

Gómez-Rodríguez, C. and Nivre, J. (2010). A Transition-Based Parser for 2-Planar Dependency Structures. In *Proceedings of the 48th Annual Meeting of the Association*

for Computational Linguistics, pages 1492–1501, Uppsala, Sweden. Association for Computational Linguistics.

Hajič, J., Ciaramita, M., Johansson, R., Kawahara, D., Martí, M. A., Màrquez, L., Meyers, A., Nivre, J., Padó, S., Štěpánek, J., Straňák, P., Surdeanu, M., Xue, N., and Zhang, Y. (2009). The CoNLL-2009 Shared Task: Syntactic and Semantic Dependencies in Multiple Languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL 2009): Shared Task*, pages 1–18, Boulder, Colorado. Association for Computational Linguistics.

Hale, J. (2001). A probabilistic Earley parser as a psycholinguistic model. In *Proceedings of the second meeting of the North American Chapter of the Association for Computational Linguistics on Language technologies*, pages 159–166. Association for Computational Linguistics.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *ACM SIGKDD explorations newsletter*, 11(1):10–18.

Hancke, J., Vajjala, S., and Meurers, D. (2012). Readability Classification for German using Lexical, Syntactic, and Morphological Features. In *Proceedings of COLING 2012*, pages 1063–1080, Mumbai, India. The COLING 2012 Organizing Committee.

Hassan, H., Sima'an, K., and Way, A. (2009). Lexicalized Semi-Incremental Dependency Parsing. In *Proceedings of the Recent Advances in NLP (RANLP'09)*, Borovets, Bulgaria. Association for Computational Linguistics.

Hays, D. (1964). Dependency Theory: A Formalism and Some Observations. *Language*, 40:511–525.

Hockenmaier, J. (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*, pages 505–512, Sydney, Australia. Association for Computational Linguistics.

Hockenmaier, J. and Steedman, M. (2002). Generative Models for Statistical Parsing with Combinatory Categorial Grammar. In *Proceedings of 40th Annual Meeting of*

the Association for Computational Linguistics, pages 335–342, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Hockenmaier, J. and Steedman, M. (2007). CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Honnibal, M. and Curran, J. R. (2007). Improving the complement/adjunct distinction in CCGBank. *Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics (PACLING-07)*, pages 210–217.

Honnibal, M., Curran, J. R., and Bos, J. (2010). Rebanking CCGbank for Improved NP Interpretation. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 207–215, Uppsala, Sweden. Association for Computational Linguistics.

Honnibal, M., Goldberg, Y., and Johnson, M. (2013). A Non-Monotonic Arc-Eager Transition System for Dependency Parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172, Sofia, Bulgaria. Association for Computational Linguistics.

Honnibal, M., Nothman, J., and Curran, J. R. (2009). Evaluating a statistical CCG parser on Wikipedia. In *Proceedings of the 2009 Workshop on The People's Web Meets NLP: Collaboratively Constructed Semantic Resources*, pages 38–41. Association for Computational Linguistics.

Huet, G. (1975). A unification algorithm for typed -calculus. *Theoretical Computer Science*, 1(1):27 – 57.

Husain, S. (2009). Dependency Parsers for Indian Languages. In *Proceedings of the ICON09 NLP Tools Contest: Indian Language Dependency Parsing*, India.

Husain, S., Mannem, P., Ambati, B. R., and Gadde, P. (2010). The ICON-2010 Tools Contest on Indian Language Dependency Parsing. In *Proceedings of ICON-2010 Tools Contest on Indian Language Dependency Parsing*, Kharagpur, India.

Hwang, W., Hajishirzi, H., Ostendorf, M., and Wu, W. (2015). Aligning Sentences from Standard Wikipedia to Simple Wikipedia. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguis-*

tics: Human Language Technologies, pages 211–217, Denver, Colorado. Association for Computational Linguistics.

Iida, R., Komachi, M., Inui, K., and Matsumoto, Y. (2007). Annotating a Japanese text corpus with predicate-argument and coreference relations. In *Proceedings of the Linguistic Annotation Workshop*, pages 132–139. Association for Computational Linguistics.

Joachims, T. (2006). Training linear SVMs in linear time. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 217–226. ACM.

Johansson, R. and Nugues, P. (2007). Extended constituent-to-dependency conversion for English. In *16th Nordic Conference of Computational Linguistics*, pages 105–112. University of Tartu.

Joshi, A., Vijay-Shanker, K., and Weir, D. (1991). The Convergence of Mildly Context-Sensitive Formalisms. In Sells, P., Shieber, S., and Wasow, T., editors, *Processing of Linguistic Structure*, pages 31–81. MIT Press, Cambridge, MA.

Katz-Brown, J., Petrov, S., McDonald, R., Och, F., Talbot, D., Ichikawa, H., Seno, M., and Kazawa, H. (2011). Training a Parser for Machine Translation Reordering. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 183–192, Edinburgh, Scotland, UK. Association for Computational Linguistics.

Kawahara, D., Kurohashi, S., and Hasida, K. (2002). Construction of a Japanese Relevance-tagged Corpus. In *LREC*.

Kim, S. M., Ng, D., Johnson, M., and Curran, J. (2012). Improving Combinatory Categorial Grammar Parse Reranking with Dependency Grammar Features. In *Proceedings of COLING 2012*, pages 1441–1458, Mumbai, India. The COLING 2012 Organizing Committee.

Kiparsky, P. and Staal, J. F. (1969). Syntactic and semantic relations in Pāini. *Foundations of Language*, pages 83–117.

Klein, D. and Manning, C. D. (2003). Accurate Unlexicalized Parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan. Association for Computational Linguistics.

Kübler, S., McDonald, R., and Nivre, J. (2009). *Dependency Parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Kumari, B. and Rao, R. R. (2015). Improving Telugu Dependency Parsing using Combinatory Categorial Grammar Supertags. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 14(1):3.

Kuperman, V., Stadthagen-Gonzalez, H., and Brysbaert, M. (2012). Age-of-acquisition ratings for 30,000 English words. *Behavior Research Methods*, 44(4):978–990.

Kwiatkowski, T., Choi, E., Artzi, Y., and Zettlemoyer, L. (2013). Scaling Semantic Parsers with On-the-Fly Ontology Matching. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1545–1556, Seattle, Washington, USA. Association for Computational Linguistics.

Kwiatkowski, T., Zettlemoyer, L., Goldwater, S., and Steedman, M. (2010). Inducing Probabilistic CCG Grammars from Logical Form with Higher-Order Unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1223–1233, Cambridge, MA. Association for Computational Linguistics.

Lafferty, J., McCallum, A., and Pereira, F. C. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289.

Lewis, M., He, L., and Zettlemoyer, L. (2015). Joint A* CCG Parsing and Semantic Role Labelling. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1444–1454, Lisbon, Portugal. Association for Computational Linguistics.

Lewis, M. and Steedman, M. (2013a). Combined Distributional and Logical Semantics. *Transactions of the Association for Computational Linguistics*, 1:179–192.

Lewis, M. and Steedman, M. (2013b). Unsupervised Induction of Cross-Lingual Semantic Relations. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 681–692, Seattle, Washington, USA. Association for Computational Linguistics.

Lewis, M. and Steedman, M. (2014a). A* CCG Parsing with a Supertag-factored Model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, Doha, Qatar. Association for Computational Linguistics.

Lewis, M. and Steedman, M. (2014b). Improved CCG parsing with Semi-supervised Supertagging. *Transactions of the Association for Computational Linguistics (TACL)*, 2:327–338.

Magerman, D. M. (1994). *Natural Language Parsing as Statistical Pattern Recognition*. PhD thesis, Stanford University.

Mahajan, A. (2000). Relative Asymmetries and Hindi Correlatives. In Alexiadou, A., Law, P., Meinunger, A., and Wilder, C., editors, *The Syntax of Relative Clauses*, pages 201–229. Amsterdam: John Benjamins.

Mannem, P., Chaudhry, H., and Bharati, A. (2009). Insights into non-projectivity in Hindi. In *Proceedings of the ACL-IJCNLP 2009 Student Research Workshop*, pages 10–17, Suntec, Singapore. Association for Computational Linguistics.

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Marslen-Wilson, W. (1973). Linguistic structure and speech shadowing at very short latencies. *Nature*, 244:522–533.

Martins, A., Smith, N., and Xing, E. (2009). Concise Integer Linear Programming Formulations for Dependency Parsing. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 342–350, Suntec, Singapore. Association for Computational Linguistics.

McDonald, R. (2006). *Discriminative learning and spanning tree algorithms for dependency parsing*. PhD thesis, Philadelphia, PA, USA.

McDonald, R., Crammer, K., and Pereira, F. (2005a). Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 91–98, Ann Arbor, Michigan. Association for Computational Linguistics.

McDonald, R. and Nivre, J. (2007). Characterizing the Errors of Data-Driven Dependency Parsing Models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131, Prague, Czech Republic. Association for Computational Linguistics.

McDonald, R., Nivre, J., Quirmbach-Brundage, Y., Goldberg, Y., Das, D., Ganchev, K., Hall, K., Petrov, S., Zhang, H., Täckström, O., Bedini, C., Bertomeu Castelló, N., and Lee, J. (2013). Universal Dependency Annotation for Multilingual Parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia, Bulgaria. Association for Computational Linguistics.

McDonald, R., Pereira, F., Ribarov, K., and Hajic, J. (2005b). Non-Projective Dependency Parsing using Spanning Tree Algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 523–530, Vancouver, British Columbia, Canada. Association for Computational Linguistics.

Meyers, A., Reeves, R., Macleod, C., Szekely, R., Zielinska, V., Young, B., and Grishman, R. (2004). The NomBank Project: An Interim Report. In Meyers, A., editor, *HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, pages 24–31, Boston, Massachusetts, USA. Association for Computational Linguistics.

Mikolov, T. (2012). *Statistical Language Models Based on Neural Networks*. PhD thesis, Brno University of Technology.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient Estimation of Word Representations in Vector Space. In *Proceedings of Workshop at ICLR*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013b). Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc.

Miyao, Y. and Tsujii, J. (2008). Feature forest models for probabilistic HPSG parsing. *Computational Linguistics*, 34(1):35–80.

Mnih, A. and Hinton, G. (2009). A Scalable Hierarchical Distributed Language Model. In *Advances in Neural Information Processing Systems*, volume 21, pages 1081–1088.

Mohanan, K. P. (1982). Grammatical relations in Malayalam. *In Joan Bresnan (ed.), The Mental Representation of Grammatical Relations*.

Mohanan, T. (1994). Argument Structure in Hindi. *CSLI Publications*.

Nivre, J. (2003). An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*, pages 149–160.

Nivre, J. (2004). Incrementality in Deterministic Dependency Parsing. In Keller, F., Clark, S., Crocker, M., and Steedman, M., editors, *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain. Association for Computational Linguistics.

Nivre, J. (2008). Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Nivre, J. (2009). Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Suntec, Singapore. Association for Computational Linguistics.

Nivre, J., Hall, J., Kübler, S., McDonald, R., Nilsson, J., Riedel, S., and Yuret, D. (2007a). The CoNLL 2007 Shared Task on Dependency Parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, Prague, Czech Republic. Association for Computational Linguistics.

Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., and Marsi, E. (2007b). MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

Nivre, J. and Nilsson, J. (2005). Pseudo-Projective Dependency Parsing. In *ACL '05: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 99–106, Ann Arbor, Michigan. Association for Computational Linguistics.

Nocedal, J. and Wright, S. (1999). *Numerical optimization*. Springer.

Ouchi, H., Duh, K., and Matsumoto, Y. (2014). Improving Dependency Parsers with Supertags. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers*, pages 154–158, Gothenburg, Sweden. Association for Computational Linguistics.

Palmer, M., Kingsbury, P., and Gildea, D. (2005). The Proposition Bank: An Annotated Corpus of Semantic Roles. *Computational Linguistics*, 31(1):71–106.

Pareschi, R. and Steedman, M. (1987). A Lazy way to Chart-Parse with Categorial Grammars. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 81–88, Stanford, California, USA. Association for Computational Linguistics.

Petersen, S. E. and Ostendorf, M. (2009). A machine learning approach to reading level assessment. *Computer speech & language*, 23(1):89–106.

Petrov, S. and Klein, D. (2007). Improved Inference for Unlexicalized Parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 404–411, Rochester, New York. Association for Computational Linguistics.

Pyysalo, S., Ginter, F., Heimonen, J., Björne, J., Boberg, J., Järvinen, J., and Salakoski, T. (2007). BioInfer: a corpus for information extraction in the biomedical domain. *BMC bioinformatics*, 8(1):50.

Quirk, C. and Menezes, A. (2006). Do we need phrases? Challenging the conventional wisdom in Statistical Machine Translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 9–16, New York City, USA. Association for Computational Linguistics.

Reddy, S., Lapata, M., and Steedman, M. (2014). Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics*, 2:377–392.

Reddy, S., Täckström, O., Collins, M., Kwiatkowski, T., Das, D., Steedman, M., and Lapata, M. (2016). Transforming Dependency Structures to Logical Forms for Semantic Parsing. *Transactions of the Association for Computational Linguistics*, 4.

Roark, B. (2001). Probabilistic Top-Down Parsing and Language Modeling. *Computational Linguistics*, 27:249–276.

Robinson, J. (1970). Dependency Structures and Transformational Rules. *Language*, 46:259–285.

Sagae, K., Miyao, Y., and Tsujii, J. (2007). HPSG Parsing with Shallow Dependency Constraints. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 624–631, Prague, Czech Republic. Association for Computational Linguistics.

Sangati, F. and Keller, F. (2013). Incremental Tree Substitution Grammar for Parsing and Sentence Prediction. In *Transactions of the Association for Computational Linguistics (TACL)*.

Sartorio, F., Satta, G., and Nivre, J. (2013). A Transition-Based Dependency Parser Using a Dynamic Parsing Strategy. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 135–144, Sofia, Bulgaria. Association for Computational Linguistics.

Schwartz, L., Callison-Burch, C., Schuler, W., and Wu, S. (2011). Incremental Syntactic Language Models for Phrase-based Translation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 620–631, Portland, Oregon, USA. Association for Computational Linguistics.

Sennrich, R. (2015). Modelling and Optimizing on Syntactic N-Grams for Statistical Machine Translation. *Transactions of the Association for Computational Linguistics*, 3:169–182.

Sharma, D. M., Mannem, P., vanGenabith, J., Devi, S. L., Mamidi, R., and Parthasarathi, R., editors (2012). *Proceedings of the Workshop on Machine Translation and Parsing in Indian Languages*. The COLING 2012 Organizing Committee, Mumbai, India.

Shastri, C. (1973). Vyakarana Chandrodya (Vol. 1 to 5). *Delhi: Motilal Banarsidass. (In Hindi)*.

Siddharthan, A. and Mandya, A. (2014). Hybrid text simplification using synchronous dependency grammars with hand-written and automatically harvested rules. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 722–731, Gothenburg, Sweden. Association for Computational Linguistics.

Socher, R., Bauer, J., Manning, C. D., and Andrew Y., N. (2013). Parsing with Compositional Vector Grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 455–465, Sofia, Bulgaria. Association for Computational Linguistics.

Steedman, M. (2000). *The Syntactic Process*. MIT Press, Cambridge, MA, USA.

Stoness, S. C., Tetreault, J., and Allen, J. (2004). Incremental Parsing with Reference Interaction. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 18–25.

Tanenhaus, M., Spivey-Knowlton, M., Eberhard, K., and Sedivy, J. (1995). Integration of visual and linguistic information in spoken language comprehension. *Science*, 268(5217):1632–1634.

Tratz, S. and Hovy, E. (2011). A Fast, Accurate, Non-Projective, Semantically-Enriched Parser. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1257–1268. Association for Computational Linguistics.

Tse, D. and Curran, J. R. (2010). Chinese CCGbank: extracting CCG derivations from the Penn Chinese Treebank. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 1083–1091, Beijing, China. Coling 2010 Organizing Committee.

Tse, D. and Curran, J. R. (2012). The Challenges of Parsing Chinese with Combinatory Categorial Grammar. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 295–304, Montréal, Canada. Association for Computational Linguistics.

Turian, J., Ratinov, L.-A., and Bengio, Y. (2010). Word Representations: A Simple and General Method for Semi-Supervised Learning. In *Proceedings of the 48th Annual*

Meeting of the Association for Computational Linguistics, pages 384–394, Uppsala, Sweden. Association for Computational Linguistics.

Uematsu, S., Matsuzaki, T., Hanaoka, H., Miyao, Y., and Mima, H. (2013). Integrating Multiple Dependency Corpora for Inducing Wide-coverage Japanese CCG Resources. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1042–1051, Sofia, Bulgaria. Association for Computational Linguistics.

Uematsu, S., Matsuzaki, T., Hanaoka, H., Miyao, Y., and Mima, H. (2015). Integrating Multiple Dependency Corpora for Inducing Wide-Coverage Japanese CCG Resources. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 14(1):1–24.

Vaidya, A., Husain, S., Mannem, P., and Sharma, D. M. (2009). A karaka-based dependency annotation scheme for English. In *Proceedings of Computational Linguistics and Intelligent Text Processing (CICLing)*, pages 41–52.

Vajjala, S. (2015). *Analyzing text complexity and text simplification: connecting linguistics, processing and educational applications*. PhD thesis, University of Tübingen, Germany.

Vajjala, S. and Meurers, D. (2014). Assessing the relative reading level of sentence pairs for text simplification. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 288–297, Gothenburg, Sweden. Association for Computational Linguistics.

Wang, W. and Harper, M. (2003). Language Modeling Using a Statistical Dependency Grammar Parser. In *Proceedings of the International Workshop on Automatic Speech Recognition and Understanding*, US Virgin Islands.

Watanabe, T. and Sumita, E. (2015). Transition-based Neural Constituent Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1169–1179, Beijing, China. Association for Computational Linguistics.

Weiss, D., Alberti, C., Collins, M., and Petrov, S. (2015). Structured Training for Neural Network Transition-Based Parsing. In *Proceedings of the 53rd Annual Meeting of*

the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 323–333, Beijing, China. Association for Computational Linguistics.

Wilson, M. (1988). MRC Psycholinguistic Database: Machine-usable dictionary, version 2.00. *Behavior Research Methods, Instruments, & Computers*, 20(1):6–10.

Woodsend, K. and Lapata, M. (2011). WikiSimple: Automatic Simplification of Wikipedia Articles. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI)*, pages 927–932, San Francisco, California, USA.

Wubben, S., van den Bosch, A., and Krahmer, E. (2012). Sentence Simplification by Monolingual Machine Translation. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1015–1024, Jeju Island, Korea. Association for Computational Linguistics.

Xu, W., Auli, M., and Clark, S. (2015a). CCG Supertagging with a Recurrent Neural Network. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 250–255, Beijing, China. Association for Computational Linguistics.

Xu, W., Callison-Burch, C., and Napoles, C. (2015b). Problems in Current Text Simplification Research: New Data Can Help. *Transactions of the Association for Computational Linguistics*, 3:283–297.

Xu, W., Clark, S., and Auli, M. (2016). Shift-Reduce CCG Parsing with Recurrent Neural Networks and Expected F-Measure Training. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, San Diego, California. Association for Computational Linguistics.

Xu, W., Clark, S., and Zhang, Y. (2014). Shift-Reduce CCG Parsing with a Dependency Model. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 218–227, Baltimore, Maryland. Association for Computational Linguistics.

Xue, N., Xia, F., Chiou, F.-D., and Palmer, M. (2005). The Penn Chinese TreeBank: Phrase structure annotation of a large corpus. *Natural language engineering*, 11(02):207–238.

Yamada, H. and Matsumoto, Y. (2003). Statistical Dependency Analysis with Support Vector Machines. In *Proceedings of the Eighth International Workshop on Parsing Technology*, pages 195–206.

Zhang, H., Huang, L., Zhao, K., and McDonald, R. (2013). Online Learning for Inexact Hypergraph Search. In *Proceedings of the conference on Empirical methods in natural language processing*.

Zhang, Y. and Clark, S. (2008). A Tale of Two Parsers: Investigating and Combining Graph-based and Transition-based Dependency Parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 562–571, Honolulu, Hawaii. Association for Computational Linguistics.

Zhang, Y. and Clark, S. (2011a). Shift-Reduce CCG Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 683–692, Portland, Oregon, USA. Association for Computational Linguistics.

Zhang, Y. and Clark, S. (2011b). Syntactic Processing Using the Generalized Perceptron and Beam Search. *Computational Linguistics*, 37:105–151.

Zhang, Y. and Nivre, J. (2011). Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA. Association for Computational Linguistics.

Zhang, Y. and Nivre, J. (2012). Analyzing the Effect of Global Learning and Beam-Search on Transition-Based Dependency Parsing. In *Proceedings of COLING 2012: Posters*, pages 1391–1400, Mumbai, India. The COLING 2012 Organizing Committee.

Zhou, H., Zhang, Y., Huang, S., and Chen, J. (2015). A Neural Probabilistic Structured-Prediction Model for Transition-Based Dependency Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1213–1222, Beijing, China. Association for Computational Linguistics.

Zhu, Z., Bernhard, D., and Gurevych, I. (2010). A Monolingual Tree-based Translation Model for Sentence Simplification. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 1353–1361, Beijing, China. Coling 2010 Organizing Committee.