Lambek Categorial Grammars for Practical Parsing

by

Timothy Alexander Dalton Fowler

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Lambek Categorial Grammars for Practical Parsing

Timothy Alexander Dalton Fowler

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2016

This dissertation investigates which grammar formalism to use for representing the structure of natural language in a practical parser, and advocates the use of Lambek Categorial Grammar (LCG) over similar formalisms such as Combinatory Categorial Grammar (CCG).

Before we argue for the advantages of LCG, we first overcome two obstacles to its use in practical parsers: its NP-Complete parsing problem, and its weak equivalence to Context-Free Grammars (CFGs). We develop a parsing algorithm for LCG that is polynomial when the order of categories in the grammar is bounded by a constant. Furthermore, we show that in CCGbank, the only existing categorial grammar corpus, the order of categories is very low. Next, we analyze the Clark and Curran parser, the state of the art in categorial grammar parsing, and establish that the CCG that it uses is also weakly equivalent to CFGs. Then, we train the Petrov parser, a probabilistic CFG parser, on CCGbank and obtain the best results to date on the test set. This establishes that LCG's weak equivalence to CFGs is not a hindrance to its use in a practical parser.

Having established the viability of using LCG in a parser, we then argue for its superiority over CCG as a representation for the structure of natural language sentences. We show that LCG offers a greater transparency between its syntactic structure and the categorial semantics that can be built from a parse of a categorial grammar. As part of this argument, we show that representing LCG derivations as proof nets allows for a

more cohesive representation of categorial syntax, dependency structures and categorial semantics than CCG. To demonstrate this, we develop a corpus of LCG derivations by semi-automatically translating the CCG derivations of CCGbank into LCG proof nets.

Dedication

To my grandfather, Stanley Maurice Raymond, who showed me that there is always more to learn.

Acknowledgements

First, I would like to thank my supervisor, Gerald Penn, for his help throughout my graduate studies. Without his constant help and advice, I would not have even known where to start.

I would also like to thank the rest of my committee, Graeme Hirst and Fahiem Bacchus. Their feedback during the process of writing this dissertation was invaluable. Also, many thanks goes to my external examiner, Stephen Clark, who provided a lot of the inspiration behind this dissertation.

Also, I want to thank all of my friends and colleagues who accompanied me through this PhD. This includes all of the people in the computational linguistics research group who helped keep me working towards my goal, and my friends who helped distract me from it.

And, my family, to which I will be forever grateful for supporting me. My grandparents for encouraging me to stick with it, my parents for never questioning why I would want to and my brother for showing me that I'm not the only one.

And, finally, to my beautiful wife, who became part of my life along with this PhD, but who will remain with me long after it is over.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation will advocate the use of Lambek Categorial Grammar (LCG) (Lambek, 1958) as a grammar formalism for practical parsing. Our arguments take several distinct forms. We will present an efficient parsing algorithm that is more efficient than the well-known parsing algorithm for Combinatory Categorial Grammar (CCG) (Steedman, 2000), under certain conditions. Then, we will perform an analysis of the weak generative capacity of the Clark and Curran parser (Clark and Curran, 2007b), which is the current state of the art in practical categorial grammar parsing, and demonstrate that LCG is as expressive as the grammar that the Clark and Curran parser is based on. Finally, we will perform an analysis of CCGbank (Hockenmaier and Steedman, 2007), a CCG corpus, and show that this corpus has a number of deficiencies that LCG can correct. As part of this, we develop a corpus of LCG derivations and show that LCG provides an improvement in the transparency between syntax and compositional semantics.

1.1 Overview

At its heart, this dissertation investigates what formal systems, or grammar formalisms, should be used to specify the structure of natural language in a practical parser. Our focus will be on categorial grammars (Ajdukiewicz, 1935, Bar-Hillel, 1953), and we will

contrast categorial grammars with each other and also with other grammar formalisms, such as Context-Free Grammars (CFGs) (Chomsky, 1957) and dependency grammars (Tesnière, 1959). Among the categorial grammars, we will specifically advocate the use of LCG and show that it has a number of advantages, most particularly over CCG.

The attributes of a grammar formalism that will be of primary interest will be those related to practical parsing. In particular, this means a focus on a few things:

1. The efficiency of parsing
2. The generative capacity of the formalism
3. The ease with which we can build representations such as semantic terms and dependency structures from the syntactic representation

In terms of efficiency, our goal will be to provide a polynomial-time parsing algorithm. In terms of the language class of a grammar, we want to evaluate the string language that a grammar generates, or in other words, locate its position on the Chomsky hierarchy. In more technical terms, efficiency becomes a question of the asymptotic complexity of a parsing algorithm, while generative capacity is often expressed as the string languages that can be generated by the grammar formalism, also known as its *weak generative capacity*. In chapters 3 and 4, we will address each of these topics in turn and argue that LCG is sufficient for practical parsing in terms of efficiency and the language class of a grammar.

Practical Parsing

Part of what distinguishes the work presented in this dissertation is that we will focus on *practical* parsing. That is, we will be focusing on the aspects of grammar formalisms that are important for constructing natural language corpora of real-world sentences, like the Penn Treebank (Marcus et al., 1994) and CCGbank (Hockenmaier and Steedman,

2007), and for building wide-coverage parsers that train on those corpora, such as the Collins parser (Collins, 1999), the Charniak parser (Charniak, 2000), the Stanford parser (Klein and Manning, 2003), the Petrov parser (Petrov et al., 2006) and the Clark and Curran parser (Clark and Curran, 2007b). Each of these parsers is a statistical parser that trains on a corpus of real-world sentences and outputs a syntactic representation, which has proven to be useful for various natural language processing tasks.

A central theme in our advocacy of LCG over other categorial grammar formalisms used in practical parsers is a return to the core principles of categorial grammar. In particular, this means that a categorial grammar should be *strongly lexicalized*, or, in other words, it should delegate nearly all of the grammar to the lexicon rather than to the rule system. In addition, we will argue for the importance of the *categorial semantics* of a categorial grammar derivation and, in particular, the requirement that there be a clear relationship between the syntactic and semantic derivations of a categorial grammar. As we will see in chapter 5, the non-categorial rules of CCGbank and the Clark and Curran parser contravene both of these conditions.

A practical natural language parser is a computational system that generates structures for natural language sentences or utterances. These structures typically take the form of a syntactic analysis of the sentence. However, this syntactic analysis is often used to generate a semantic representation, because semantic representations have proven to be useful in a number of natural language processing tasks (Bos and Markert, 2005, de Marneffe et al., 2006a, McCarty, 2007). Our primary argument for the use of LCG over competing formalisms like CFGs or CCG is that LCG provides syntactic derivations that allow for more straightforward construction of semantic terms. We will refer to the ease of constructing semantic terms from syntactic representations as the *transparency* between the syntax and the semantics of a formalism. We will not introduce a formal or mathematical measure of transparency partly because such a measure would be difficult to define, and partly because such a measure will not be necessary to see the advantages

of our LCG derivations over their CFG or CCG counterparts for a sentence.

The crucial difference between LCG and CCG is that the LCG rule system is fixed, whereas each particular CCG has a list of combinators that specifies which rules can appear in any syntactic representation. If this list of combinators is kept small and follows certain principles, semantic counterparts can be defined for the syntactic combinators allowing for a semantic term to be constructed in a principled way alongside the syntactic representation. However, the Clark and Curran parser and CCGbank contain a large number of rules for which defining semantic counterparts is cumbersome, and require that ad hoc and unprincipled methods be applied. As a result, it is difficult to ascertain the quality of the semantic terms built from the syntactic output of the Clark and Curran parser and the syntactic structures in CCGbank. In chapter 5, we will develop a corpus that adheres to these core categorial grammar principles, allowing for a clear and verifiable construction of semantic terms from the syntactic derivations in the corpus and evaluate it.

Generative Capacity

One of the subtopics that we will address in this dissertation is that of the generative capacity of grammar formalisms. There has been a great deal of interest in both the languages of strings and the languages of trees necessary for capturing all of natural language. The study of the tree languages of grammar formalisms gives us insight into the types of natural language structures that can be represented by a grammar formalism, but the results in this area are limited by a variety of theoretical factors. Therefore, researchers have focused most of their attention on the string languages generated by grammar formalisms, also known as their weak generative capacity. The idea is that if a grammar formalism is not powerful enough to generate the necessary string languages to represent natural language, then it cannot possibly be powerful enough to generate the necessary tree languages.

There have been two particularly important results on the weak generative capacity of grammar formalisms for practical parsing. The first is that natural language is not even weakly context free because certain languages contain cross-serial dependencies (Shieber, 1985, Culy, 1985). The second is that four independently developed grammar formalisms that can generate such cross-serial dependencies all have the same weak generative capacity (Joshi et al., 1991). One of those formalisms, CCG, has been the foundation of a research program that has culminated in the development of: (1) the Clark and Curran parser (Clark and Curran, 2007b), a CCG parser that competes with the state of the art in practical parsing, and (2) CCGbank (Hockenmaier and Steedman, 2007), the CCG corpus upon which it trains. In chapter 4, we will investigate the extent to which both the Clark and Curran parser and CCGbank are non-context-free. We will show that the Clark and Curran parser is strongly context-free, or rather, that it generates only tree languages that a Context-Free Grammar can generate. In addition, we will investigate the structures found in CCGbank, and we will show that CCGbank does not contain the types of rules necessary for a CCG to generate non-context-free languages. Conversely, LCG is well-known to be weakly context-free (Pentus, 1997), which has been used as an argument against its usefulness as a practical parsing formalism. However, these results show that the context-free property of LCG is not a hindrance relative to the state of the art in practical parsing.

Three Types of Linguistic Structure

State-of-the-art parsers often maintain multiple representations of structure for a sentence which may include three types: phrase-structure trees, dependency structures and semantic terms.

Both the CFG-based and the CCG-based parsers maintain a phrase-structure representation that groups words into phrases, labels the phrases and then specifies a hierarchy among phrases.

In addition to phrase structure, it has often been found useful to generate dependency structures for a sentence. Dependency structures specify relationships between pairs of words in a sentence, which can be labelled or unlabelled. Dependency structures serve two purposes in the parsing literature: (1) as the syntax of a parser (McDonald et al., 2005), or (2) as an additional syntactic representation of the parser, alongside a phrase-structure representation (Clark et al., 2002, Clark and Curran, 2007b, de Marneffe et al., 2006b).

Semantic terms are a kind of structure that can be assigned to a sentence that represents the semantics or meaning of the sentence. Semantic terms that can be found in the parsing literature take a variety of forms, including logical forms (Montague and Thomason, 1974) and discourse representation structures (Kamp and Reyle, 1993).

Our primary argument advocating of the use of LCG in practical wide-coverage parsing over other categorial grammars, such as CCG, is the cohesion of phrase-structure trees, dependency structures and semantic terms within LCG. The syntax of an LCG derivation has two representations that are closely related: its representation as a logical proof, and its representation as a proof net. The former is a close counterpart of the phrase-structure trees of CCG and CFG, while the latter is more closely related to dependency structures, especially those found in the CCG literature (Clark et al., 2002). Furthermore, because LCG is a formalism that is strongly grounded in logic, its derivations have a semantics that are easily obtained from its syntactic representations.

Due to the large number of combinatory rules in practical wide-coverage CCG parsers, obtaining dependency structures or semantic terms from the syntactic representation of CCG is not straightforward. In principle, the CCG combinators are all derivatives of the combinators of combinatory logic, but in practice a large number of non-categorial rules appear in CCGbank, and consequently, the Clark and Curran parser. In particular, there are currently three different systems for generating dependencies from a CCG parse: one used to generate the dependency structures included in CCGbank (Hockenmaier and

Steedman, 2007), one used to generate the dependency structures inside the Clark and Curran parser (Clark and Curran, 2007b), and one developed independently by Stephen Boxwell (Boxwell, 2010). Each of these systems generates identical dependencies for CCG parses containing only categorial rules, but differ in their treatment of non-categorial rules. The only system currently available to generate semantics from a CCG parse is Boxer (Bos et al., 2004), but it suffers from the same essential problem as the dependency generators, in that it has problems dealing with non-categorial rules.

Variants of Lambek Categorial Grammar

There are a number of variants of LCG in the literature. Lambek's original definition (Lambek, 1958) includes the product connective and bans empty premises. In this dissertation, we will only consider LCG without product because product is not necessary linguistically and it adds a significant degree of complexity, especially in developing our parsing algorithm in chapter 3. The first definition of LCG without product was given by Cohen (1967). With respect to empty premises, it is irrelevant whether these are allowed in determining parsing complexity and weak generative capacity. Chapter 3 considers both LCG allowing empty premises and banning empty premises, while chapter 5 does not use empty premises in any of its derivations.

1.2 Thesis Statement

The thesis statement of this dissertation consists of two parts: the first concerning the viability of LCG as a practical parsing formalism, and the second concerning the greater transparency between syntax and semantics in LCG.

Thesis Statement. Lambek Categorial Grammar is viable for practical parsing, and there is greater transparency between its syntax and semantics than other grammar formalisms used in practical parsing.

LCG was introduced as a grammar formalism over 50 years ago (Lambek, 1958), and was long suspected of having an NP-complete parsing problem and being weakly equivalent to Context-Free Grammars. The latter was proven two decades ago by Pentus (1997), and the former was proven more recently by Savateev (2008). Both of these results have been used as arguments for why LCG cannot be used as a practical grammar formalism. Chapters 3 and 4 will respond to each of these issues in turn, arguing for the viability of LCG as a practical grammar formalism.

In chapter 3, we will give a parsing algorithm for LCG that is polynomial under certain conditions, and we will show that those conditions are met by the derivations found in CCGbank. Although we don't currently have full formal proofs, we will provide partial proofs and intuitive arguments for the remainder. In chapter 4, we will show that the current state of the art in practical parsing is context-free, by performing an analysis of the Clark and Curran parser (Clark and Curran, 2007b) and showing that despite using CCG as its basis, the Clark and Curran parser is not using the non-context-free portions of that formalism.

The second part of our thesis states that LCG is superior to similar grammar formalisms, such as CFG and CCG, due to the transparency between its syntax and semantics. In particular, the inherent simplicity and inflexibility of the rule system of LCG allows a principled construction of semantic terms and dependency structures from the syntactic representation of LCG. Our arguments for the superior transparency of its syntax-semantics interface will take the form of an analysis of the current state of the art in categorial grammar corpora, CCGbank (Hockenmaier and Steedman, 2007), and the development of an LCG corpus constructed by semi-automatically converting the derivations found in CCGbank. We will look at a number of linguistic constructions found in CCGbank, including relative clauses, wh-movement, coordination and sentential gapping, among others. We will show that although the analyses in CCGbank have a syntactic simplicity, the construction of both semantic terms and dependency structures

from the syntactic structures found in CCGbank is necessarily convoluted and unprincipled. We will then provide LCG derivations for these constructions that allow for a principled construction of semantic terms from the syntactic structures of LCG.

1.3 Contributions

The contributions of this dissertation fall into three distinct areas: theoretical results concerning parsing with LCG, theoretical and empirical results concerning the generative power of CCG, and empirical results regarding LCG derivations on sentences from the Penn Treebank. More specifically, these contributions can be described as:

- The first efficient algorithm for parsing with LCG, albeit without complete proofs (chapter 3)
- The first demonstration of the context-freeness of the Clark and Curran parser (chapter 4)
- The first corpus of categorial grammar derivations that strictly adheres to the principles of categorial grammar (chapter 5)

Other minor contributions include the following:

- A simpler characterization of the proof nets of Penn (2004) (section 3.2)
- A framework that can be used to define the various definitions of CCG (section 4.1.1)
- A characterization of the set of rules found in CCGbank (section 4.1.3)
- A method for training a probabilistic Context-Free Grammar parser on CCGbank (section 4.2)

- A demonstration of the ineffectiveness of the features found in CCGbank (section 4.2.3)
- Dependency parsing results that are competitive with the state of the art for sentences from CCGbank (section 4.2.4)
- The development of the first corpus of proof nets (chapter 5)

1.4 The Structure of the Document

This dissertation is structured into three distinct arguments. The first two argue for the viability of LCG as a practical parsing formalism and the last argues that LCG provides better access to semantics than the alternatives.

Chapter 2 offers a review of the background literature on categorial grammar. This includes technical introductions to AB Grammar (ABG), also known as Classical Categorial Grammar, (section 2.3), Steedman's definition of CCG (section 2.4) and LCG (section 2.5). Furthermore, this includes brief overviews of CCGbank and the Clark and Curran parser (section 2.4.1), and an introduction to the proof nets of Penn (2004) (section 2.5.3).

Chapter 3 presents an algorithm for parsing with LCG. Included in that chapter is a simplified version of proof nets based on those of Penn (2004) (section 3.2) and a discussion of order, which is a measure of a category's complexity (section 3.6). We also present partial proofs of the algorithm's correctness, along with intuitive arguments for the remainder and an analysis of its computational complexity. Our argument for correctness is reinforced by our implementation of this algorithm that is discussed in section 5.6.

Chapter 4 presents our analysis of the generative power of CCG. That chapter begins with our formal definition of a CCG (section 4.1), then proceeds with our framework for specifying the various definitions of CCG from the literature (section 4.1.1), and

proves that the definitions of CCG in practice are strongly context-free (sections 4.1.2 and 4.1.3). We then proceed to discuss the use of the Petrov parser (Petrov and Klein, 2007) on CCGbank (section 4.2), including four evaluation methods of the Petrov parser against the Clark and Curran parser according to supertags (section 4.2.2), PARSEVAL measures (section 4.2.3), dependencies (section 4.2.4), and time and space (section 4.2.5).

Chapter 5 introduces our argument that LCG provides greater transparency between syntax and semantics than CCG. We offer a number of problematic linguistic constructions in CCGbank, and introduce LCG solutions to these constructions that allow for a compositional semantics and a dependency structure to be constructed in a principled way. We finish with our method for automatically introducing these new derivations into CCGbank, outline our method for building a corpus of proof nets from the phrase-structure trees in CCGbank and then evaluate the new corpus.

Finally, chapter 6 offers some conclusions and future directions.

Chapter 2

Categorial Grammars

Categorial grammars are a family of grammar formalisms that can be characterized as having a rich lexicon where syntactic categories and semantic terms for words are specified as functions. A lexical entry for a word consists of a syntactic category and a semantic term that are in close correspondence. This leads to the primary advantage of using categorial grammars, where each syntactic parse is accompanied by a semantic parse yielding a semantic term for the sentence. These semantic terms can take a variety of forms, such as first-order logic terms and discourse representation structures, and have been used for inference in a natural language understanding system (Bos et al., 2004, Bos and Markert, 2005).

2.1 Fundamentals

The functions in the syntactic lexicon differ from the usual definition of functions by specifying both the type of argument and the *direction* of the argument. The types of arguments are specified from a set of atoms, and the direction of arguments are specified using two connectives[1]: \ and /. In both cases, the argument is to the right of the slash

[1]The \ and / are the most common connectives from a large family found in the categorial grammar literature.

and the result is to the left. The difference between the two symbols is that \ requires arguments to be to the left and / requires arguments to be to the right[2].

In addition to our two symbols for expressing arguments, we require a set of atoms over which the functions can be defined. The set that is typically used in the literature consists of a subset of the usual parts of speech from classical linguistics: $\{S, NP, N, PP\}$, corresponding to the parts of speech for sentence, noun phrase, noun and prepositional phrase. A notable omission from this set is any category for verb phrases, since verb phrases are defined as functions over the atoms, in categorial grammar.

To demonstrate this notation, a transitive verb like "loves" is typically assigned the following category:

(2.1) $(S\backslash NP)/NP$

An alternative orthography for the category $(S\backslash NP)/NP$ is $S\backslash NP/NP$. In more formal terms, we will assume left-associativity for / and \. Based on our definitions above, this category specifies a transitive verb as a function requiring a noun phrase to its right, then a noun phrase to its left. Then, the result, or return value, is a sentence.

Alongside each syntactic category in the lexicon, categorial grammars also specify a semantic term for the word. Since the syntactic half of the grammar specifies the directions of arguments, the semantic half need not. The undirected equivalent of the categorial grammar syntax is the lambda calculus, which is used to represent the compositional component of semantics in categorial grammar.

To do this, we need the function-defining symbol λ, an infinite set of variables, $\{x, y, z, \ldots\}$, and some set of semantic atoms. The set of semantic atoms will vary depending on which semantic theory one adheres to, and can include more sophisticated symbols like quantifiers and feature structures, if necessary. In this dissertation,

[2]This definition of \ and / is referred to as Steedman notation (Steedman, 2000). Ajdukiewicz notation switches the two operands of the \ and is the notation commonly found in the theoretical categorial grammar literature (Lambek, 1958, Carpenter, 1998, Morrill, 2010).

we will use constants corresponding to words, some logical connectives such as $\wedge$, for conjunction, and some quantifiers such as $\exists$ for existential quantification, $\exists!$, for unique existential quantification and $\forall$ for universal quantification.

For example, one might use the set $\{mary, jon, loves\}$ for a simple grammar together with the logical connectives $\neg$ and $\wedge$. The semantic term for the transitive verb "loves" would then be the following:

(2.2) $\lambda xy.love(y, x)$

These examples exhibit the correspondence between the syntactic categories and semantic terms in the sense that both categories require the same number of arguments. This correspondence is a fundamental principle of categorial grammar, and is what allows categorial grammars to build semantic terms in a principled way.

Each categorial lexicon contains some number of categories that can be assigned to entire sentences. We will use the symbol τ to denote the symbol in the lexicon that maps to the categories for sentences.

We can now generally define the categorial grammar parsing problem. The inputs to the parsing problem are a sentence and a lexicon, which we will refer to as a sentence-lexicon pair. The problem then asks whether there is an assignment of categories to words such that the sequence of categories is derivable from one of the categories in the lexicon for τ, according to the rules of the categorial grammar.

Having now defined our lexicon and the general parsing problem, we need to define how these functions can be combined into syntactic and semantic structures for whole sentences. The specific rules allowed differ depending on the particular variant of categorial grammar. We will discuss a number of different categorial grammars, but first, we need a notion of the complexity of a category.

2.2 The Order of a Category

Before we proceed with discussing the various categorial grammars, we will first introduce the concept of the *order* of a category. The order of a category is an important method for measuring a category's complexity.

Informally, the notion of the order of a category is the *depth* of the nesting of its arguments. In particular, a category that takes only atomic categories as arguments, such as the category $S\backslash NP/NP$, has order 1. A category where its arguments include a category of order 1, such as $S\backslash NP/(S\backslash NP)$, has order 2. Due to the fact that we use Steedman notation and we assume left-associativity and minimal bracketing, the order of a category in this dissertation is the number of nested parentheses plus one.

More formally, order can be defined as follows:

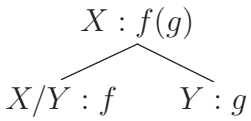
Definition 2.2.1. Given a category c, its *order* $o(c)$ is defined recursively as follows:

- $o(c) = 0$ if c is atomic
- $o(c) = max(o(a), o(b) + 1)$ for $c = a/b$ or $c = a\backslash b$

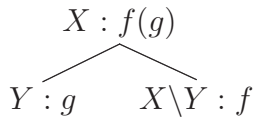
As far as we are aware, the first definition of order for categories was Aarts (1994). Order will be important because the complexity of the parsing algorithm in chapter 3 is exponential in the order of the categories in the input. In section 3.6, we will analyze the order of categories in CCGbank and discover that the order of categories never exceeds 5, and only rarely exceeds 3.

2.3 AB Grammar

The simplest variant of categorial grammar consists of a set of rules corresponding to function application. This variant of categorial grammar is the basis of all other categorial



(a) Right Application.



(b) Left Application.

Figure 2.1: The application rules of ABG.

grammars and is known as AB Grammar (ABG) (Ajdukiewicz, 1935, Bar-Hillel, 1953)[3]. Because of the directionality encoded into arguments, a *right* and a *left* application are required.

The rules of ABG are shown in figure 2.1, and specify both how the syntactic categories are combined and how the semantic terms are combined. The syntactic categories appear to the left of the : and the semantic terms appear to the right. The rule of Right Application can be read as follows. If we have two adjacent phrases, where the left phrase has syntactic category X/Y and the right phrase has syntactic category Y, for some X and Y, then we may apply the *Right Application* rule of ABG to obtain the syntactic category X for the entire string of words. Then, if f is the semantic term for the phrase with category X/Y and g is the semantic term for the phrase with category Y, the semantic term for the entire string is $f(g)$, or rather, the term obtained by applying f to g. The *Left Application* rule functions similarly except in a leftward direction.

An important distinction between the application rules presented above and the typical rules found in a CFG is that the application rules are actually templates for rules similar to those found in CFGs. The X and Y symbols are variables that can be instantiated to any category, thereby generating an infinite set of rule instances. We will consider this subject more carefully in chapter 4.

[3]In the literature, AB grammar is also referred to as classical categorial grammar or pure categorial grammar.

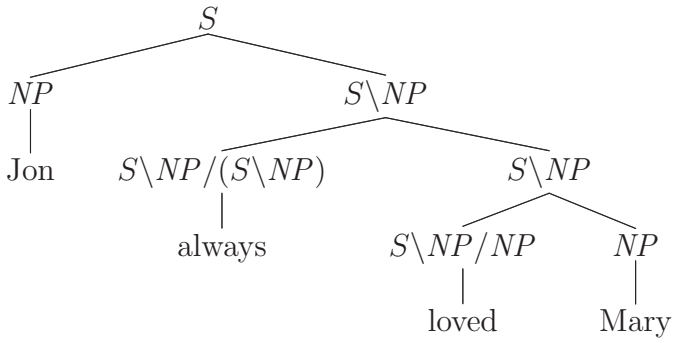


Figure 2.2: An ABG syntactic derivation for "Jon always loved Mary".

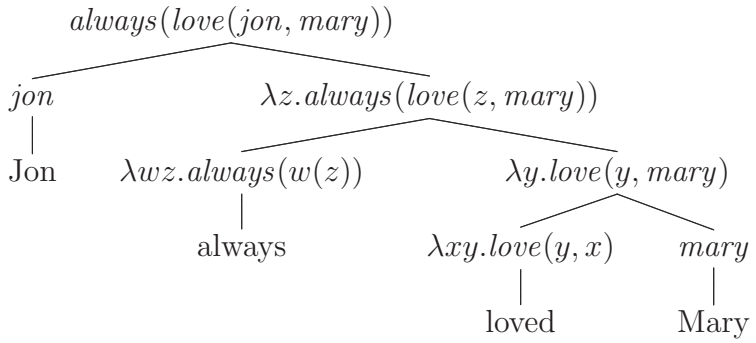


Figure 2.3: The semantics for the ABG derivation in figure 2.2.

ABG is well understood from a theoretical perspective. The parsing problem for ABG is polynomial time (Bar-Hillel, 1953) and its weak generative capacity is the class of context-free languages (Bar-Hillel et al., 1960). An example syntactic derivation is shown in figure 2.2 and the accompanying semantic derivation is shown in figure 2.3.

ABG in particular, and categorial grammar more broadly, is situated between phrase-structure grammars (such as CFGs) and logical inference systems (such as classical logic linear logic). On the one hand, one can think of a categorial grammar as a kind of phrase-structure grammar. However, instead of enumerating a finite list of rules, as is the case with CFGs, categorial grammars allow templates of rules to effectively provide an infinite list of rules allowed in the grammar. Cast in this framework, the lexicon is simply a finite list of rules assigning categories to words, and a categorial grammar is

simply a constrained infinitely large phrase-structure grammar. On the other hand, one can think of a categorial grammar as a logical inference system. In this case, the rules of ABG are inference rules in our logic, and the lexicon enumerates the logical objects, or sequences of categories, which must be derived according to the inference rules. Such a view of categorial grammar allows us to treat our grammar as a logic, and gives us access to tools from the logic community that will be particularly important for handling Lambek Categorial Grammar in section 2.5. In the literature, this view is referred to as "parsing as deduction".

The primary problem with ABG as a practical formalism is that it cannot handle any of the extraction phenomena that are commonly found in natural language. In particular, it has no way of modelling syntactic structures such as wh-movement and relative clauses. The primary shortcoming is that noun phrases representing the arguments of certain verbs in a sentence are not found in the expected position, but are found elsewhere in the sentence, with the result that functional application cannot occur. Sections 2.4 and 2.5 introduce Combinatory Categorial Grammar and Lambek Categorial Grammar, respectively, each of which offers a different solution to this expressivity problem.

2.4 Combinatory Categorial Grammar

Combinatory Categorial Grammar (CCG) (Joshi et al., 1991, Steedman, 2000, Clark and Curran, 2007b, Hockenmaier and Steedman, 2007) extends ABG by adding combinatory rules to the application rules of ABG. Combinatory rules can take a wide variety of forms, but they are generally motivated by some combination of linguistics and combinatory logic (Curry and Feys, 1958). The combinatory rules combine syntactic categories in more sophisticated ways than functional application, while generally obeying the fact that the categories are functions. For example, the composition rules in figures 2.4a and 2.4b are linguistically necessary to analyze relative clauses and wh-movement, but also

correspond to functional composition, which is an inference rule in combinatory logic.

As a result of requiring a collection of rules, CCG does not have a single specific definition, but rather a number of related definitions depending on the requirements of the author. Chapter 4 details a variety of definitions of CCG and investigates the differences between them. In this section, we will present the definition of CCG found in Steedman (2000) for expository purposes.

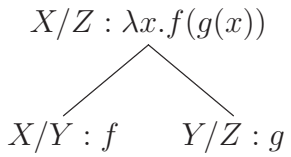
The rules of Steedman CCG (Steedman, 2000) are given in figure 2.4, and two new notations are introduced. First, $|_i$ is used to indicate a variable over the set $\{\backslash, /\}$[4] and the index is used to reference a given variable in multiple locations. Second, the symbol ... specifies an alternating sequence of categories and slashes, where the slashes are constrained according to the adjacent slashes in the obvious way.

The rules of figure 2.4, together with the application rules of ABG and a categorial lexicon, specify a grammar formalism that has more expressive power than ABG alone. In particular, the composition rules can be used to model various extraction phenomena, like relative clauses and wh-questions, and the substitution rules can be used to model parasitic gapping constructions. Without any other additions, this grammar formalism is referred to as *Pure CCG* (Kuhlmann et al., 2010). However, Steedman CCG allows a grammar to specify certain restrictions on the use of the rules in CCG, as follows:

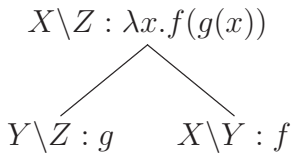
- Each of the rules in a CCG can be restricted to a finite set of rule instances, where some of the variables have been replaced by categories.

The intuition behind these rule restrictions is to allow a grammar to, for example, restrict the use of crossed composition to only the case where $X = S\backslash NP$. This way of restricting rules is necessary for this definition of Steedman CCG to generate the same string languages as tree-adjoining grammars (Kuhlmann et al., 2010).

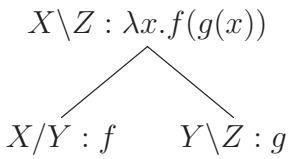
[4]The use of the $|_i$ notation is only to present the rules more concisely. All rules with an instance of $|_i$ could be replaced by two rules obtained by replacing $|_i$ by each of / or \.



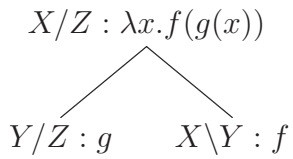
(a) Forward Composition.



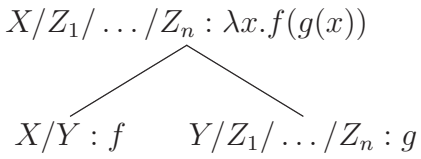
(b) Backward Composition.



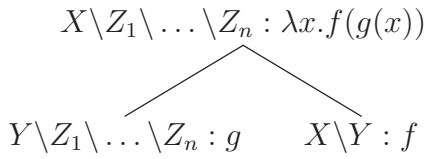
(c) Forward Crossed Composition.



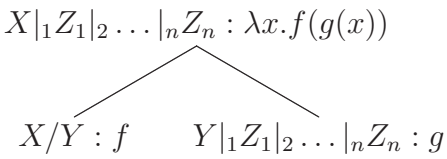
(d) Backward Crossed Composition.



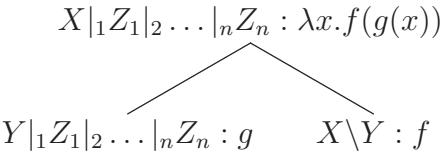
(e) Generalized Forward Composition.



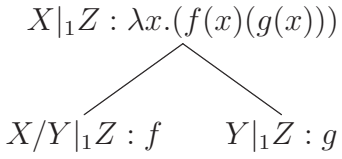
(f) Generalized Backward Composition.



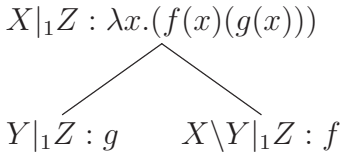
(g) Generalized Forward Crossed Composition.



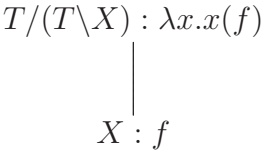
(h) Generalized Backward Crossed Composition.



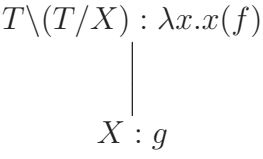
(i) Forward Substitution.



(j) Backward Substitution.



(k) Forward Restricted Type-Raising.



(l) Backward Restricted Type-Raising.

Figure 2.4: The rules of Steedman's Combinatory Categorial Grammar.

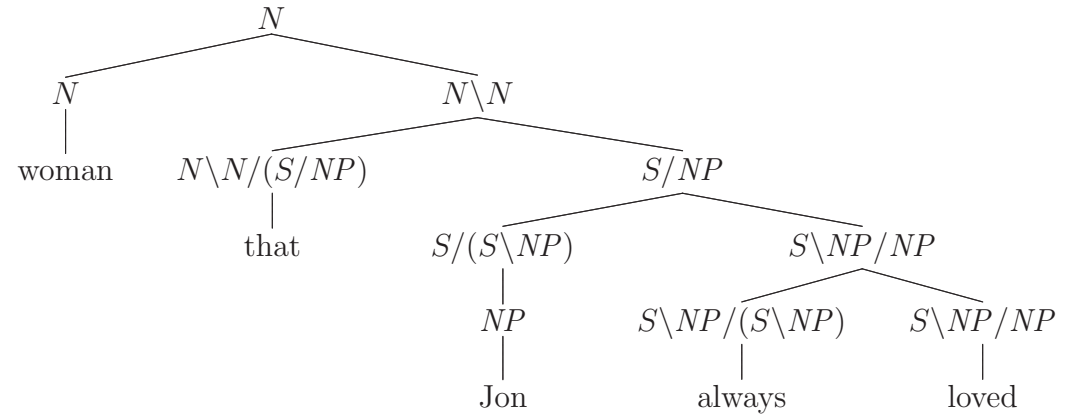


Figure 2.5: A CCG syntactic derivation for a relative clause.

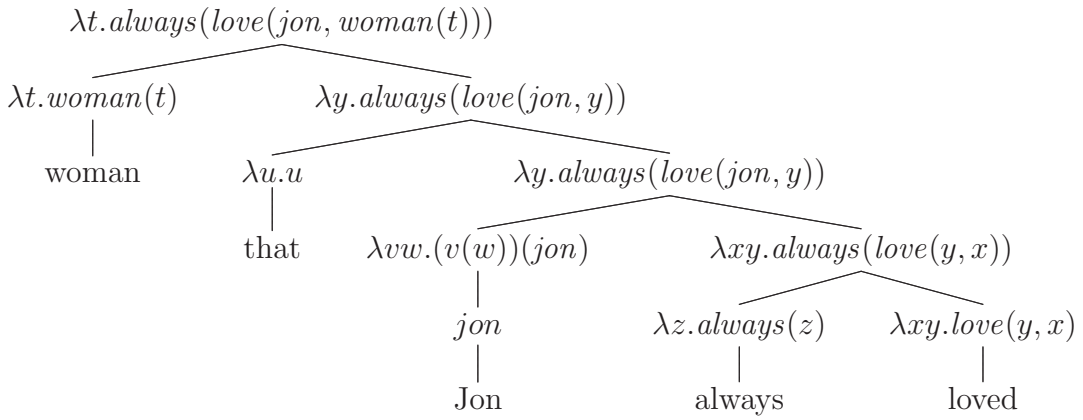


Figure 2.6: The CCG semantic derivation for the syntactic derivation in figure 2.5.

A simple example of a CCG parse that accounts for the syntax of a relative clause is shown in figure 2.5, and its semantic counterpart is shown in figure 2.6. The CCG composition rules are necessary to combine the categories for "always" and "loved". The type-raising rule is necessary to allow the category $S\backslash NP/NP$ to apply to its leftward argument before its rightward argument. As can be seen, type-raising provides partial associativity for CCG.

A more complex CCG derivation and the corresponding semantics are shown in figures 2.7 and 2.8, respectively. The words "without" and "reading" are combined using an instance of composition, and the word "file" and the phrase "without reading" are

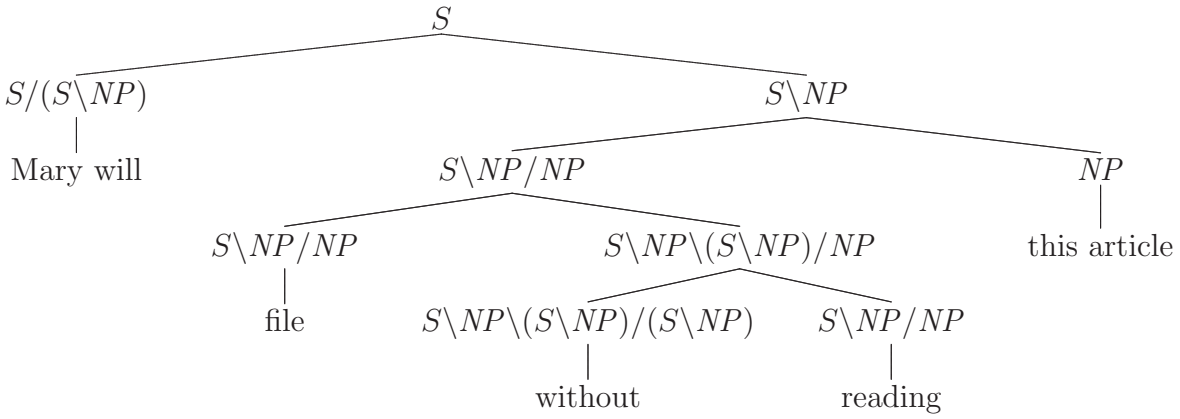


Figure 2.7: A CCG example derivation for "Mary will file without reading this article", adapted from Steedman (2000).

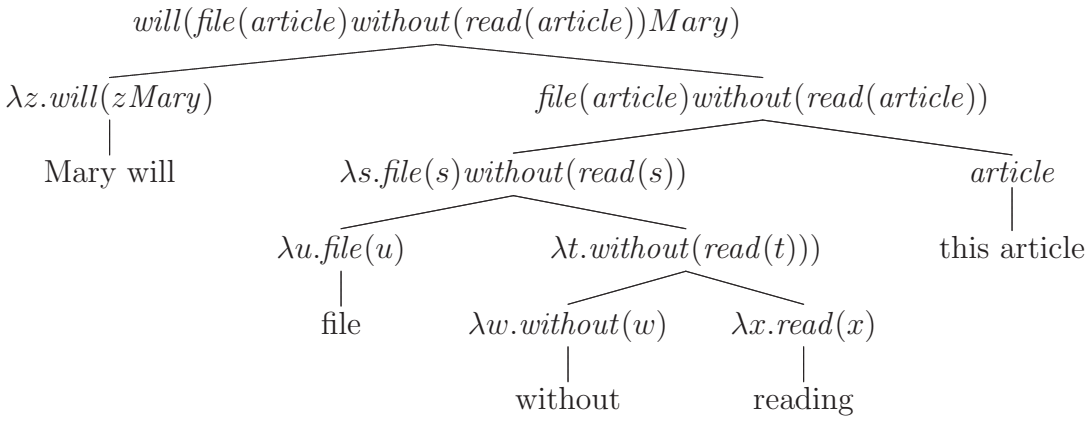


Figure 2.8: The semantics for the CCG derivation in figure 2.7.

combined using an instance of substitution. The semantic rule for substitution allows a single variable to appear twice in the term, although we will see in chapter 5 that this can be accomplished without such rules by increasing the complexity of the lexicon.

Steedman CCG is theoretically well-understood, and is known to have a polynomial time parsing algorithm (Vijay-Shanker and Weir, 1990, Kuhlmann and Satta, 2014) and to be weakly equivalent to tree-adjoining grammars (Vijay-Shanker and Weir, 1994). In particular, CCG can generate the string language $\{a^n b^n c^n d^n | n \geq 1\}$, which can be used to analyze the cross-serial dependencies of Swiss German (Shieber, 1985).

2.4.1 Practical CCG

One of the primary strengths of CCG is that it is well-established as both a theoretical formalism and as a practical formalism. Its theoretical properties, such as the complexity of its parsing problem (Vijay-Shanker and Weir, 1990) and its weak generative capacity (Joshi et al., 1991), are well understood. It has also had great success in the field of parsing, with the development of the Clark and Curran parser (Clark and Curran, 2007b) and CCGbank (Hockenmaier and Steedman, 2007).

Recently, the field of parsing has had a revolution in terms of its movement towards statistical methods. This began with the development of the Penn Treebank (Marcus et al., 1994), and the subsequent development of statistical context-free grammar parsers (Collins, 1999, Charniak, 2000).

However, Context-Free Grammars have two major flaws, in terms of representing the meaning of natural language. First, there is no clear relationship between the syntactic representations found in the Penn Treebank and any kind of compositional semantic representation. Second, it is well-known that Context-Free Grammars cannot generate cross-serial dependencies found in languages such as Swiss German and Dutch (Shieber, 1985). As a result, a CCG corpus, known as CCGbank (Hockenmaier and Steedman, 2007), and a wide-coverage CCG parser, known as the Clark and Curran parser (Clark and Curran, 2004, Bos et al., 2004), were developed. The former contains CCG derivations (and also dependency structures) of the sentences in the Wall Street Journal section of the Penn Treebank[5]. The latter is a statistical parser that uses a discriminative probability model trained on CCGbank and generates syntactic derivations. These syntactic derivations can then be fed into the semantic backend, called Boxer, that converts the syntactic derivations to one of several types of semantic term.

[5]275 of the sentences in the Penn Treebank were left out of CCGbank purportedly because CCG cannot handle the linguistic constructions in those sentences.

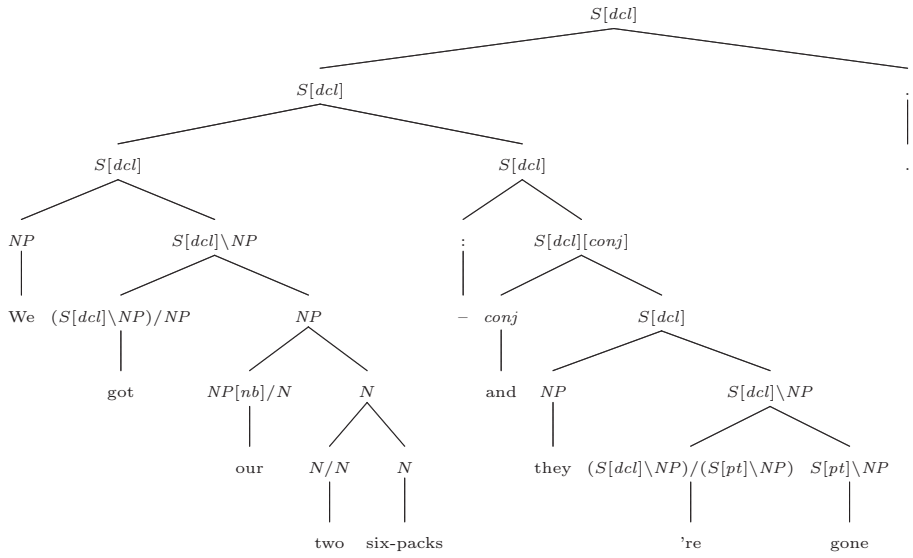


Figure 2.9: The CCGbank phrase-structure tree for sentence 0071.38.

CCGbank

Prior to the development of the Penn Treebank (Marcus et al., 1994), wide-coverage statistical parsers were not possible because no corpus annotated with syntactic derivations existed that was large enough to use as training data. Similarly, prior to the development of CCGbank, the same could be said of wide-coverage statistical CCG parsers.

However, independent of any parser, CCGbank is important because it is the only resource we have that contains real-world sentences along with categorial grammar derivations. This allows statistical analyses to be done on real-world categorial grammar derivations, such as determining the size of a practical categorial lexicon and frequency of various linguistic phenomena classified by their analysis within categorial grammar.

The development of CCGbank began with the phrase-structure trees found in the Penn Treebank and then underwent a series of transformations. First, the derivations in the Penn Treebank were binarized, and then, from among the children for each binary rule, a head was chosen based on the head-finding heuristics of Magerman (1994) and Collins (1999). Then, the heads were assigned a function category that takes each of its

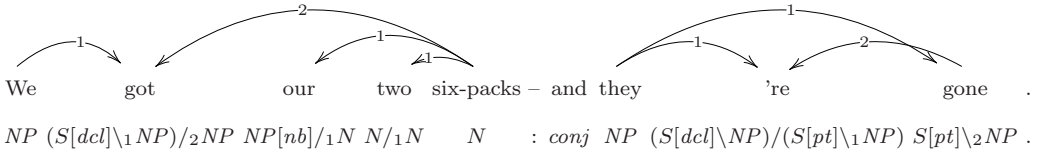


Figure 2.10: The CCGbank dependency structure for sentence 0071.38.

siblings as arguments. An example CCG derivation from CCGbank is shown in figure 2.9.

Alongside the CCG derivations in CCGbank, CCGbank also provides dependency structures for each sentence. Dependency theory is a theory of syntax and semantics, originally developed by Tesnière (1959), that is an alternative to the phrase-structure derivations that we have presented thus far. A dependency structure over a sequence of words is a directed acyclic graph over the words, where the edges may or may not have labels.

The dependency structure for the sentence shown in figure 2.9 is shown in figure 2.10. The dependencies are shown above the words in the sentence and the syntactic categories are shown below. The dependencies of CCGbank are labelled according to which argument they correspond to among the syntactic categories. Dependencies are drawn from the dependent to the head. For example, the word "got" has a category that takes two arguments, both of which are of category NP. The innermost argument is the subject of "got", which is represented by the dependency labelled by 1 from the word "We" to the word "got". Its outermost argument is the phrase "our two six-packs", which is represented by the dependency from the word "six-packs" to the word "got" labelled by 2.

One should first notice the rules in the CCG derivation in figure 2.9 that do not resemble any of the categorial rules that we have discussed so far. In particular, the rules for punctuation and the rule for coordination resemble their phrase-structure equivalents

from the Penn Treebank. These rules are part of a much more widespread issue in CCGbank that involves non-categorial rules for a number of phenomena in the corpus. The primary problem with such rules is how they affect construction of dependency structures and semantic terms. To handle them requires definitions of the counterpart semantic rules to generate the semantic terms, but there is no theoretical basis to turn to for such rules. We will discuss these issues at length in chapter 5.

This problem also manifests itself in the conversion from CCG derivations to dependency structures in CCGbank, and in the Clark and Curran parser. The conversion of CCG derivations to dependency structures has been established in the theoretical literature (Clark et al., 2002). To obtain a dependency structure over words, that theory essentially assigns a dependency whenever a functional application occurs. Then, additional information from a lexicon is used to reorient some of these edges locally. However, in practice, because of the large number of non-categorial rules in CCGbank, this process is not fully defined. In particular, there are currently three systems in the literature used for the conversion of CCGbank derivations to dependency structures:

1. Hockenmaier's original system that generated the dependencies in CCGbank
2. The subprogram `generate` included with the Clark and Curran parser
3. Stephen Boxwell's PARG Generator (Boxwell, 2010)

The primary problem is that these all produce somewhat different dependency structures for a given CCG derivation, and even their coverage varies by up to five percent on sentences from CCGbank. This is compounded by the fact that Hockenmaier's original system for generating CCGbank dependencies is currently not available, and therefore we have been unable to analyze this system.

Despite these issues, it appears that dependencies are a lightweight way of representing semantics in natural language because of the strong correspondence between the dependency structure found in figure 2.10 and the logical form found in figure 2.8. In

fact, we can reconstruct logical form in its entirety with only the semantic lexicon and the information contained in a dependency structure.

The Clark and Curran Parser

The Clark and Curran Parser (Clark and Curran, 2007b) is a statistical CCG parser that trains on CCGbank. In addition to producing both CCG derivations and dependency structures as output, it can be coupled with Boxer (Bos et al., 2004), a semantic backend, that generates discourse representation structures (Kamp and Reyle, 1993) as semantic terms.

The Clark and Curran parser is notable for two reasons. First, it is the state of the art in categorial grammar parsing in terms of both speed and accuracy, and as a result its output has been used as part of an inference engine in the PASCAL Recognising Textual Entailment Challenge (Bos and Markert, 2005). The performance of that system has been quite good relative to the competition (Bar-Haim et al., 2006). Second, it is fifteen to twenty times faster than the counterpart statistical CFG parsers trained on the Penn Treebank. The superior performance of the Clark and Curran parser has been attributed to engineering and supertagging, among other factors, although the precise source has not yet been conclusively shown.

2.5 Lambek Categorial Grammar

Lambek categorial grammar (LCG) (Lambek, 1958) is another variant in the family of categorial grammar that extends ABG. LCG uses the notion of hypothetical reasoning from classical logic to handle the sorts of extraction phenomena, such as relative clauses and wh-movement, found in natural language. We will introduce hypothetical reasoning first by example and formalize it afterward.

Consider the relative clause “woman that Jon always loved” from figure 2.5. The

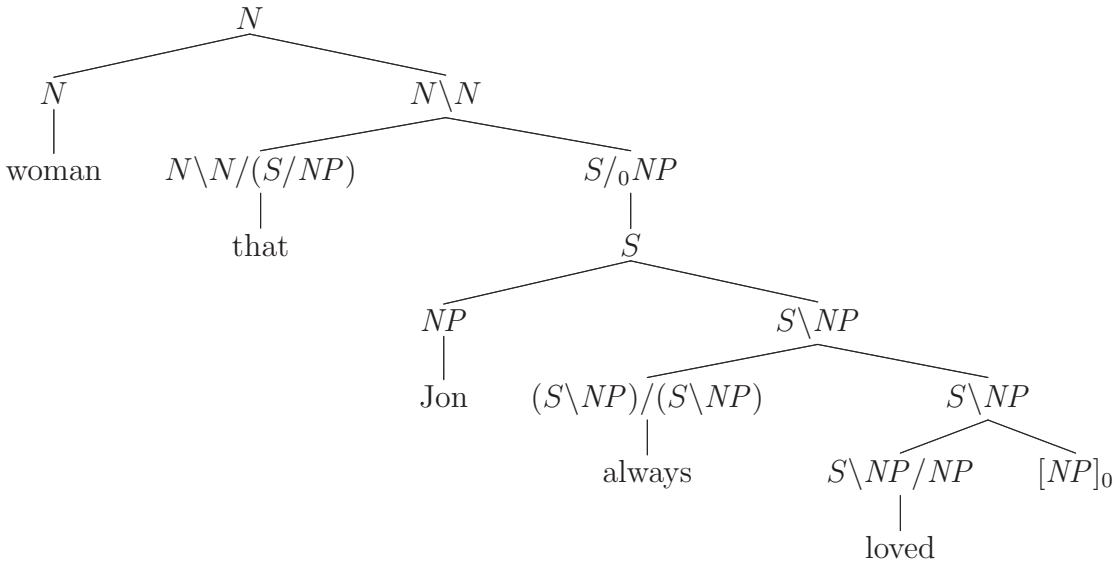


Figure 2.11: An LCG derivation for the phrase "woman that Jon always loved".

problem that ABG has is that there is no noun phrase argument to combine with the category $S\backslash NP/NP$ for "loved". Hypothetical reasoning with LCG allows us to hypothesize that a category appears at a certain position in the string. Like other forms of hypothetical reasoning, we must at some point *discharge* the hypothesis by introducing the hypothesized category along with either \ or /. The hypothesized category NP is shown in figure 2.11, along with the introduction rule introducing / with the subscript 0.

It is important that the subtree rooted at an introduction rule has the hypothesized category on either the right or the left periphery. In the former case, the discharge rule introduces a /, and in the latter, the discharge rule introduces a \. The introduction rules are shown formally in figure 2.12. The vertical ellipsis in figure 2.12 indicates an arbitrary missing portion of the proof. In figure 2.11, the missing portion of the proof is all of the rules dominating the phrase "Jon always loved". The semantic counterparts to the introduction rules use functional abstraction. They are formalized in figure 2.13, and an example is shown in figure 2.14.

We should immediately notice that the final semantic terms in figures 2.6 and 2.14 are

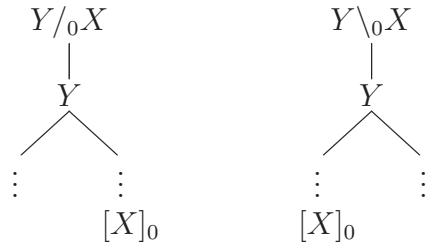


Figure 2.12: A formal presentation of LCG's introduction rules.

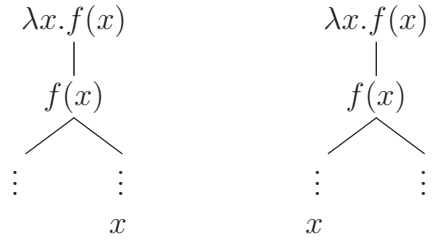


Figure 2.13: The semantic introduction rules for LCG.

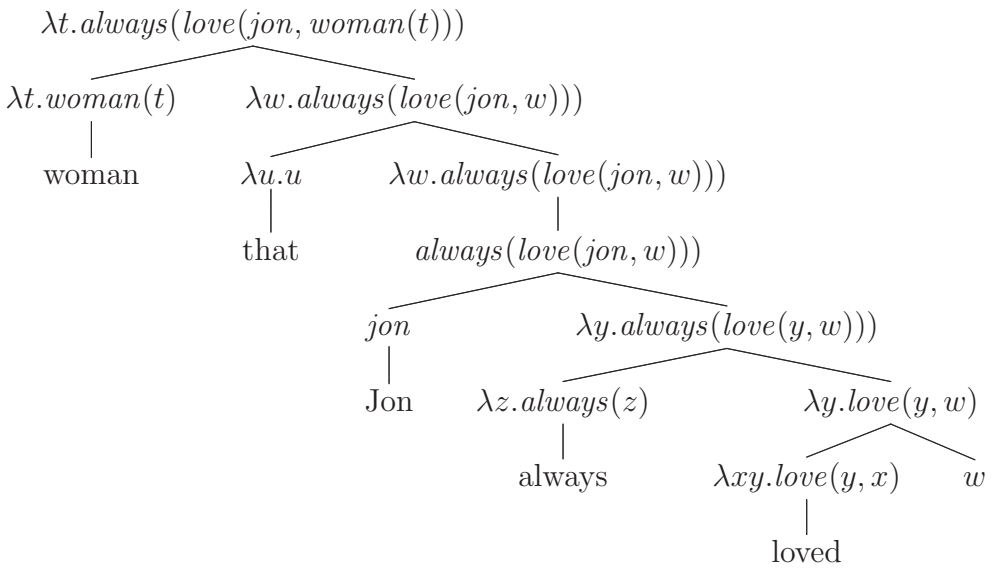


Figure 2.14: The semantic LCG derivation for the sentence in figure 2.11.

identical. This is due to the fact that both type-raising and the non-crossed composition rules of CCG are derivable in LCG. That is, we can use the introduction and application rules on the left-hand sides of those CCG rules, and build a tree with the right-hand side at the top. Furthermore, the semantics for type-raising and composition are exactly the same as the equivalent derived rules in LCG.

Given these facts, one might conclude that there is some CCG that generates exactly the same semantics as LCG, but it has been proven that the number of combinators in that CCG would necessarily be infinite (Zielonka, 1981). On the other hand, LCG cannot derive any of the crossed rules nor the substitution rules of CCG, which means that neither grammar formalism is a subset of the other. This leads us to conclude that neither LCG nor CCG is more powerful than the other in an absolute sense.

Throughout this dissertation we use the term "Lambek Categorial Grammar" or the symbol L^* to refer to the rule system defined above together with a categorial lexicon. The *Lambek Calculus* is a fragment, or subset, of linear logic (Girard, 1987) that consists of exactly the rule system of Lambek Categorial Grammar but derives sequences of categories, rather than sequences of words. In other words, we will use the term "Lambek Categorial Grammar" to refer to the grammatical system that uses the Lambek Calculus together with a lexicon that maps categories to words to provide derivations for sentences.

Pregroup grammars (Lambek, 1999) are a categorial grammar formalism that is closely related to LCG. The key difference between pregroup grammars and LCG is that pregroup grammars do not allow nesting of arguments, and, as a result, are simpler to work with. However, the similarity between CCG and pregroup grammars is not as clear as the similarity between CCG and LCG.

2.5.1 Variants of Lambek Categorial Grammar

The LCG presented above is just one grammar in a family of closely related grammars. This family of Lambek grammars can vary along the following axes:

1. Whether the grammar is associative

2. Whether the grammar includes the product connective

3. Whether the grammar allows derivations from empty premises

Non-associative LCG has been studied by a variety of authors (Aarts and Trautwein, 1995, de Groote, 1999, Moot, 2008). The primary appeal of non-associative LCG is that it has polynomial time parsing algorithms. On the other hand, the primary problem with non-associative LCG is that the input words must already be in a tree structure, which is not available from unannotated text corpora. Without such a pre-existing tree structure, all possible trees must be parsed, of which there is an exponential number.

LCG with product adds the product connective $\bullet$ to the original set of connectives $\backslash$ and $/$, and also adds relevant syntactic and semantic rules. The product connective specifies conjunction and allows categories to be conjoined within a derivation. The product connective does not increase either the computational complexity of the parsing problem or the weak generative capacity of the grammar. Furthermore, its linguistic usefulness is still an open question.

LCG that bans derivations from empty premises explicitly forbids any derivation containing a category where the leaf descendants of that category are all hypothetical. The motivation behind this variant is primarily linguistic, and stems from the fact that the ability to assign a category to a non-existent word may not be desirable. As in the case of product, allowing or banning derivations from empty premises does not change the computational complexity of the parsing problem or the weak generative capacity. At times, we will discuss LCG with empty premises banned, and we will refer to this grammar formalism as L.

In light of these variants, our original definition of LCG should more formally be referred to as the associative Lambek Categorial Grammar without product and allowing empty premises. Lambek's original grammar (Lambek, 1958) is the associative Lambek

Categorial Grammar with product and banning empty premises. We will use our definition throughout the remainder of this dissertation due to its simplicity and linguistic adequacy.

2.5.2 Algorithms for Parsing with Lambek Categorial Grammar

A number of authors have presented algorithms for parsing with LCG. König (1990) and Hepple (1992) provide chart parsing algorithms for parsing with LCG but use proofs, rather than proof nets, as their representation. Neither author does an analysis of the time complexity of their algorithm, although König (1990) states that it "has an undesirable time complexity in its general case". Furthermore, König (1990) notes that it is likely that the algorithm could be made polynomial under the right restrictions of the input space.

The first polynomial time algorithm for any fragment of LCG beyond ABG was given by Aarts (1994). Aarts provided an algorithm for deduction in the Lambek Calculus where the order of categories is at most 2. His algorithm uses the proof nets of Roorda (1991), which is proven to run in time $O(n^8)$. In addition to the rather high running time, this algorithm suffers from the problem that it does not work on categories of order 3 or more. This is problematic in practice because CCGbank contains a large number of categories with order 3 or higher.

More recently, Carpenter and Morrill (2005) provided a graph representation and a dynamic programming algorithm for parsing with LCG with product. However, their analysis lacks any reference to the running time of their algorithm, which is exponential time in the worst case.

2.5.3 Proof Nets for LCG

The hypothetical reasoning of LCG may appear to be quite difficult to handle computationally, since we would need to have some way of hypothesizing categories, and then evaluating derivations in a principled way. Fortunately, LCG is a form of linear logic (Girard, 1987), and proof theory for linear logic is well understood. Rather than analyzing actual proofs like those presented in the preceding section, Girard (1987) introduced an alternative representation: *proof nets*.

Proof nets for LCG take advantage of the fact that the lexicon in LCG is made up of functions, and the rules correspond to functional application and functional abstraction. The intuition is that proof nets match functions to arguments without specifying any kind of order among the applications and abstractions.

Proof nets represent the structure of a derivation as a graph consisting of two parts: a deterministic part representing the structure of the categories, called the *proof frame*, and a non-deterministic part representing which functions are applied to which arguments, called the *axiomatic linkage*. Together, the proof frame and the axiomatic linkage are referred to as the *proof structure*. Then, if the proof structure obeys certain conditions, it is referred to as a proof net.

Since Girard's original formulation (Girard, 1987), there have been a large number of formulations of proof nets (Danos and Regnier, 1989, Roorda, 1991, Retore, 1996, Carpenter and Morrill, 2005). We will present the formulation of Penn (2004) because of its computational tractability, which will become apparent in chapter 3. Penn's presentation of proof nets, called *LC-Graphs*, are a translation of the proof nets based on semantic terms of Roorda (1991) into a more computationally tractable graph formalism. In section 3.2 we will introduce a much simpler formulation of proof nets based on the LC-Graphs of Penn (2004).

The Proof Frame

The proof frame is a representation of the structure of the categories at the periphery of the derivation. Therefore, we need the categories across the bottom of the derivation $A_1, \ldots, A_n$ and the category at the top of the derivation B. We will refer to the categories $A_1, \ldots, A_n$ as the input categories and the category B as the output category. Sometimes we will refer to both the input and output categories in a single structure, called a *sequent*. The sequent for a sequence of input categories $A_1, \ldots, A_n$ and output category B is written $A_1, \ldots, A_n \vdash B$. Categories in proof nets receive polarities and terms: input categories receive negative polarity and the output category receives positive polarity, and each category receives a fresh variable. The intuition behind polarities is that a positive polarity is a resource and a negative polarity is a requirement of a resource, and the two need to eventually be matched up. We then decompose the input and output categories according to the following decomposition rules, which manipulate the terms and the polarities[6]:

$$
\begin{array}{lll}
(I) & (X/Y)^- : x \to X^- : xy, Y^+ : y & \\
(II) & (X\backslash Y)^- : x \to Y^+ : y, X^- : xy & \\
(III) & (X/Y)^+ : x \to Y^- : z, X^+ : y & [x := \lambda z.y] \\
(IV) & (X\backslash Y)^+ : x \to X^+ : y, Y^- : z & [x := \lambda z.y]
\end{array}
$$

Capital letters indicate categories, lower-case letters indicate terms, and y and z are fresh variables. The terms in square brackets are *side conditions*, which are generated by the positive decomposition rules. For example, the category $S\backslash NP/(S\backslash NP)^- : a$ has the following decomposition:

[6]These rules will appear somewhat different to those in the proof net literature, because we are using Steedman notation here rather than Ajdukiewicz.

$$\begin{aligned}(S\backslash NP/(S\backslash NP))^- : a &\rightarrow (S\backslash NP)^- : ab, (S\backslash NP)^+ : b \\ &\rightarrow NP^+ : c, S^- : abc, (S\backslash NP)^+ : b \\ &\rightarrow NP^+ : c, S^- : abc, S^+ : d, NP^- : e\end{aligned}$$

and generates the side condition $[b := \lambda e.d]$. We say that a variable has positive polarity if it appears as a label of a positive category anywhere in the decomposition. Otherwise, the lambda variable has negative polarity.

Each decomposition rule eliminates exactly one slash, so, after all possible decomposition rules are applied, we are left with a sequence of atoms with polarities and terms. The *proof frame* is the graph whose vertices are the polarized variables appearing in either the sequence of atoms or in one of the side conditions. Then, for each side condition $[x := \lambda z.y]$, there is an edge in the proof frame from x to y and from x to z. We will also distinguish any vertex occurring on the left side of a side condition as a *lambda-node*. The proof frame for the example above is shown in figure 2.15.

The Axiomatic Linkage

Once we have established the proof frame as a graph representing the structure of the categories, we can build an axiomatic linkage. Given the sequence of polarized atoms with terms, an axiomatic linkage requires a complete matching between positive atoms and negative atoms. Furthermore, the matching must be planar in the half plane above

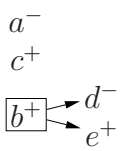
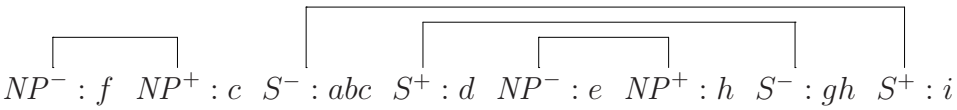


Figure 2.15: The proof frame for the category $S\backslash NP/(S\backslash NP)^- : a$ where the lambda-node b is denoted by a square box.

the atom sequence. For example, given the categories across the bottom of a derivation $NP^- : f,\ S\backslash NP/(S\backslash NP)^- : a$ and $S\backslash NP^- : g$ and the category at the top of that derivation $S^+ : i$, we get the following sequence of polarized atoms with terms:

$$NP^- : f, NP^+ : c, S^- : abc, S^+ : d, NP^- : e, NP^+ : h, S^- : gh, S^+ : i$$

Then, one possible matching is the following:



From a matching, the axiomatic linkage is the graph whose vertices are the polarized variables appearing in the sequence of atoms. Then, for each matching from a positive atom $X^+ : x$ to a negative atom $Y^- : y_1 \dots y_k$, there is an edge in the axiomatic linkage from x to y_i for $1 \leq i \leq k$. The axiomatic linkage for the matching above is shown in figure 2.16.

Proof Structures and Proof Nets

Given a sequence of input categories and an output category, its proof frame together with an axiomatic linkage for those categories is referred to as a *proof structure*. The proof structure for the axiomatic linkage shown in figure 2.16 is shown in figure 2.17. Some, but not all proof structures correspond to derivations in LCG. Those that do

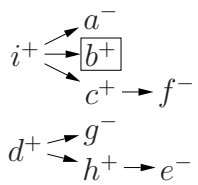


Figure 2.16: An axiomatic linkage for the categories $NP^- : f,\ S\backslash NP/(S\backslash NP)^- : a$ and $S\backslash NP^- : g$ and $S^+ : i$.

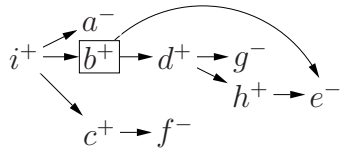


Figure 2.17: A proof structure for the proof frame built from input categories NP, $S\backslash NP/(S\backslash NP)$ and $S\backslash NP$ and output category S and the axiomatic linkage shown in figure 2.16.

correspond to derivations are referred to as *proof nets* and this class of proof structures can be precisely defined.

The following four conditions will be used to distinguish proof nets from proof structures that are not proof nets:

I(1): There is a unique vertex τ in G such that τ has in-degree 0, and for all vertices $v \in G$, there is a path from τ to v.

I(2): G is acyclic.

I(3): For every lambda-node $v \in G$, there is a path from its positive out-neighbour to its negative out-neighbour.

I(CT): For each lambda-node $v \in G$, there is a path from v to some vertex x, such that x has out-degree 0 and the last vertex before x on that path is not a lambda-node.

A proof structure G is a proof net within L^* if and only if it satisfies **I**(1-3) . A proof structure G is a proof net within L if and only if it satisfies **I**(1-3) and **I**(CT). We can easily check that the proof structure in figure 2.17 is a proof net. i^+ is the unique vertex from **I**(1), it is acyclic and the only lambda-node is $b+$, and the path $d^+ \rightarrow h^+ \rightarrow e^-$ satisfies **I**(3).

Penn (2004) proved the following theorem, which connects the notion of proof nets to the notion of proofs in LCG:

Theorem 2.5.1. *An output category can be derived in LCG from a sequence of input categories if and only if there exists a proof net over those categories.*

Proof Net Semantics

Earlier in this section, we defined semantic counterparts to the syntactic rules of LCG. However, we will be using proof nets as our syntactic representation throughout the remainder of the dissertation, so will we need a method for building semantic representations directly from a proof net for a sentence.

The term substitution used in the construction of the proof frame is actually a derivation of the semantic term for a sentence. The decomposition rules dictate the applications and abstractions that will be applied to the semantic term, and the axiomatic linkage specifies substitutions to be made.

Consider the syntactic derivation of the sentence fragment "woman that Jon always loved" in figure 2.11. The decomposition rules provide the following sequence of atoms with polarities and terms, along with the two side conditions $[c := \lambda e.f]$ and $[i := \lambda k.l]$:

$$N^- : a, N^+ : d, N^- : bcd, NP^- : e, S^+ : f, NP^- : g, NP^+ : j, S^- : hij, S^+ : l, NP^- : k, NP^+ : o, S^- : mno, NP^+ : n, N^+ : p$$

The variables assigned to the words are as follows: a is assigned to "woman", b is assigned to "that", g is assigned to "Jon", h is assigned to "always" and m is assigned to "loved". The variable for the whole sentence is p.

Then, figure 2.18 shows the LC-Graph that corresponds to the derivation in figure

2.11. This yields the following derivation of the semantic term:

$$
\begin{aligned}
& p \\
\rightarrow & ((bc)d)) \\
\rightarrow & ((b(\lambda e.f))a)) \\
\rightarrow & ((b(\lambda e.((hi)j)))a)) \\
\rightarrow & ((b(\lambda e.((h(\lambda k.l))g)))a)) \\
\rightarrow & ((b(\lambda e.((h(\lambda k.((mn)o)))g)))a)) \\
\rightarrow & ((b(\lambda e.((h(\lambda k.((me)k)))g)))a))
\end{aligned}
$$

Then, we substitute the semantic terms from the lexicon for the variables, and β-reduce:

$$
\begin{aligned}
& ((b(\lambda e.((h(\lambda k.((me)k)))g)))a)) \\
\rightarrow & ((b(\lambda e.((h(\lambda k.((\lambda xy.love(y,x)e)k)))g)))a)) \\
\rightarrow & ((b(\lambda e.((h(\lambda k.((\lambda y.love(y,e))k))g)))a)) \\
\rightarrow & ((b(\lambda e.((h(\lambda k.love(k,e))g)))a)) \\
\rightarrow & ((b(\lambda e.(((\lambda z.always(z))(\lambda k.love(k,e))g)))a)) \\
\rightarrow & ((b(\lambda e.((\lambda k.always(love(k,e)))g))a)) \\
\rightarrow & ((b(\lambda e.((\lambda k.always(love(k,e)))jon))a)) \\
\rightarrow & ((b(\lambda e.always(love(jon,e)))a)) \\
\rightarrow & (((\lambda u.u)(\lambda e.always(love(jon,e)))a)) \\
\rightarrow & ((\lambda e.always(love(jon,e)))a) \\
\rightarrow & ((\lambda e.always(love(jon,e)))\lambda t.woman(t)) \\
\rightarrow & \lambda t.always(love(jon,woman(t)))
\end{aligned}
$$

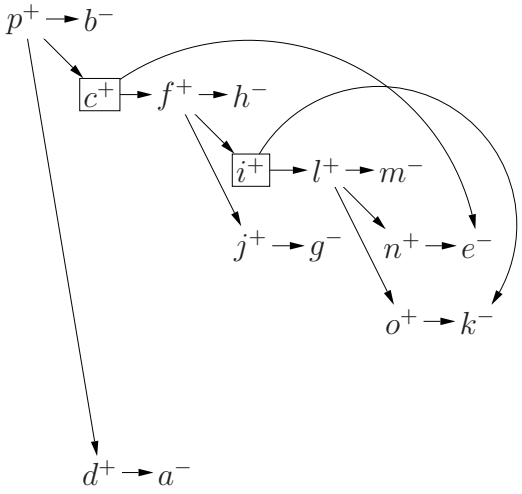


Figure 2.18: The LC-Graph corresponding to the derivation shown in figure 2.11.

And we obtain the exact semantics for the sentence given in figure 2.14.

Chapter 3

Parsing With Lambek Categorial Grammars

Given a grammar formalism and a language, the parsing problem for that grammar formalism is the problem of whether there is a structure that the grammar formalism could assign to a given sentence from the language. For categorial grammars in particular, the structure in question consists of a logical proof that takes the form of a phrase-structure tree or a proof net. Solving the parsing problem in an exponential amount of running time, relative to the size of the input, is straight-forward for all of the grammar formalisms that we have discussed so far, but an exponential running time is practically prohibitive. Therefore, our interest lies in finding parsing algorithms that run in an amount of time that is a polynomial relative to the size of the input. Without a polynomial-time parsing algorithm, a grammar formalism cannot be considered for any practical parsing task.

The literature on parsing problems for categorial grammars is mostly complete. The parsing problem for AB Grammar (ABG) was solved in the paper that introduced them (Bar-Hillel, 1953) and the parsing problem for Combinatory Categorial Grammar (CCG) was solved shortly after the introduction of CCG (Vijay-Shanker and Weir, 1990). Both of these formalisms have polynomial-time parsing algorithms; specifically, the ABG parsing

algorithm runs in time $O(n^3)$ and the CCG parsing algorithm runs in time $O(n^6)$.

The parsing problem for Lambek Categorial Grammar (LCG) stands in stark contrast to these results. A span of more than forty years passed between the introduction of LCG (Lambek, 1958) and the two results that solved the parsing problem for LCG (Pentus, 2003, Savateev, 2008). Pentus (2003) established that the parsing problem for LCG *with product* is NP-complete, and Savateev (2008) proved that the parsing problem for LCG *without product* is also NP-complete using a similar technique to the one used in Pentus's result. An NP-complete parsing problem for a grammar formalism is generally considered to be the death knell for practical parsing, but the categories that appear in these two NP-completeness results do not resemble any of the categories used in the literature on practical applications of categorial grammar (Carpenter, 1998, Steedman, 2000, Hockenmaier and Steedman, 2007). In particular, the categories in the NP-completeness proofs have arbitrarily high *order* whereas the order of categories in the linguistics literature never exceeds five and only rarely exceeds three.

In this chapter we will introduce a parsing algorithm for LCG *without product* and *allowing empty premises*[1] that is exponential time in the worst case, but which is polynomial-time when the order of categories in the input is bounded by a constant. The proof representation that we will use for LCG will be proof nets, and our algorithm will be a chart parser that uses an abstract representation of proof nets as entries. We will prove that the algorithm has a running time[2] of $O(n^4)$ when the order of categories in the input is bounded by a constant. Furthermore, we will show that our algorithm has a running time of $O(n^3)$ when each word in the lexicon is restricted to having only a constant number of category assignments and the order of the categories in the input is

[1] Generalizing the algorithm to LCG without product but banning empty premises is straightforward but adds complexity that detracts from our central point. Generalizing the algorithm to LCG *with product* using the proof nets described in Fowler (2006) is left to future research.

[2] Throughout this chapter, n will be the number of atom occurrences in the set of category occurrences for the sequence of word occurrences in the sentence plus the number of atom occurrences in the set of category occurrences for sentences. This number is strictly less than the size of the input, since the input includes both the sentence and the lexicon.

bounded by a constant.

We will not be able to provide complete proofs of correctness, but instead, we will be only be able to provide partial proofs. The remainder will consist of intuitive arguments of correctness. Theorems that would need to be proven to have complete proofs are the following: A proof that the concise representative ATG for a PTG is unique, proofs that the Bracketing and Adjoining algorithms are correct and the proof that the chart parsing algorithm is correct, which depends on the previous proofs.

Throughout this chapter we will be providing an algorithm for solving the *decision* problem for LCG, rather than trying to find all possible parses for a given sentence. If the latter is desired, one can simply specify links between entries in the chart in certain places in the algorithm to create a packed chart of parses.

This chapter is organized according to the kinds of structures necessary for representing partial states of our algorithm. Section 3.1 introduces *proof frames*, which are graphs used to represent categories that will be better than their representations as strings. In addition, proof frames make up the deterministic part of proof net construction. Then, section 3.2 introduces our version of proof nets, which are called *term graphs*. Term graphs are an improvement over the LC-Graphs of Penn (2004) because they are significantly simpler both in terms of their presentation and the number of correctness conditions. This simplicity is necessary as the algorithms that manipulate term graphs otherwise become quite complex. Next, section 3.3 introduces *partial term graphs* (PTGs), which are a representation of an incomplete term graph and which will serve as the objects that we will use to prove the correctness of the algorithm. However, PTGs will prove to be too verbose to be efficient in a chart parsing algorithm; therefore, section 3.4 introduces an abstraction over PTGs that we call *abstract term graphs* (ATGs). ATGs are the only type of graph that our algorithm will actually construct and manipulate, although the fact that they are an abstraction over PTGs will be crucial for arguing correctness. Finally, section 3.5 presents the actual chart parsing algorithm. The chart uses atom occurrences

$\mathcal{V}(G)$	The vertex set of a graph G
$\mathcal{E}(G)$	The edge set of an LC-Graph G
$\mathcal{E}_{\mathcal{R}}(G)$	The regular edge set of a graph G
$\mathcal{E}_{\mathcal{L}}(G)$	The Lambek edge set of a graph G
$\mathcal{N}(G)$	The span of a graph G
$\mathcal{O}(G)$	The order of a graph G
$\mathcal{P}(G)$	The path set of a graph G
$\mathcal{P}_{\mathcal{R}}(G)$	The regular path set of a graph G
$\mathcal{P}_{\mathcal{RS}}(G)$	The regular sibling path set of a graph G
$\mathcal{B}(a)$	Sibling node of an atom occurrence a
$\mathcal{B}^{-1}(b)$	The set of component atom occurrences of a sibling node b
$\mathcal{K}_K(a)$	Link of an atom a in linkage K
$CE(F)$	The set of crossing edges of a split in an abstract proof frame F
$CEA(F)$	$CE(F)$ along with their ancestors in the abstract proof frame F
$APF(F)$	The abstract proof frame of a proof frame F
$\mathcal{A}(c)$	The sequence of atom occurrences of a category occurrence c
$\mathcal{C}(a)$	The category occurrence of an atom occurrence a
$\mathcal{C}(w)$	The set of category occurrences of a word occurrence w
$\mathcal{W}(c)$	The word occurrence of a category occurrence c
$\mathcal{I}_{\mathcal{A}}(a)$	The index of an atom occurrence a in its category occurrence
$\mathcal{I}_{\mathcal{W}}(c)$	The index of a category occurrence c in the list of word occurrences
τ	The symbol representing the sentential categories in the lexicon

Table 3.1: Glossary of Symbols

as indices and ATGs as a representation of partial parses.

3.1 Proof Frames

The categories of an LCG are usually represented as strings consisting of atoms from the set of atoms, and the two connectives / and \. However, the underlying structure of a category is much better suited to representation by a tree, where the atoms are vertices and the connectives are expressed as edges between the atoms. This section will outline our definition of proof frames in detail, which will provide the foundation for the remainder of the chapter.

Before we begin, we must make a distinction between an atom and an atom occurrence in the usual mathematical way. In particular, we distinguish two occurrences of an atom, occurring in a single category or in different categories in a lexicon, as different *atom occurrences*. We can distinguish atom occurrences mathematically by providing each atom occurrence with a unique index, although we will refrain from doing so when such a distinction is unnecessary. Similarly, we will distinguish between categories and category occurrences, and we will often drop the term occurrence when the distinction is clear or irrelevant.

We also require the notion of polarity, due to the resource-based nature of categorial grammar. Let A be a category and let p be either positive polarity or negative polarity. We say that A^p is a *polarized category*.

The proof frame for a polarized category is a quadruple $\langle V, E_R, E_L, O\rangle$, where V is a set of vertices consisting of the atom occurrences in the category, E_R and E_L are two sets of directed edges between those vertices and O is a total order[3] over the set of vertices V. The set of *regular edges* E_R, the set of *Lambek edges* E_L and the total order O are defined via an iterative process of category decomposition. The intuition is that V, E_R and E_L are a graph representation of the argument structure of the category without direction and O specifies the directions of the arguments.

[3]Given two atom occurrences a and b and a total order over atom occurrences O, we will use the notation $a <_O b$ to indicate that a precedes b according to O.

We begin with a graph consisting of the singleton set consisting of the vertex A^p and then perform the following vertex rewriting rules:

$$(I) \qquad (\alpha/\beta)^- \Rightarrow \alpha^- \rightarrow \beta^+$$

$$(II) \qquad (\alpha\backslash\beta)^- \Rightarrow \beta^+ \leftarrow \alpha^-$$

$$(III) \qquad (\alpha/\beta)^+ \Rightarrow \beta^- \dashleftarrow \alpha^+$$

$$(IV) \qquad (\alpha\backslash\beta)^+ \Rightarrow \alpha^+ \dashrightarrow \beta^-$$

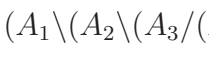
Each vertex rewriting rule specifies how to rewrite a single vertex on the left side to two vertices on the right side. The neighbourhood of the vertex on the left side of each rule is assigned to α on the right side. Dashed edges are edges in E_L and non-dashed edges are in E_R. The total order O is defined such that the category appearing on the left is $<_O$ the category appearing on the right.

This process is deterministic and since each vertex rewriting rule eliminates one slash from the graph, this process ends after a finite number of vertex rewritings terminating at categories that are atoms. Figure 3.1 shows the derivation of the proof frame for a category occurrence with negative polarity. The total order in that figure is depicted as an ordering from left to right.

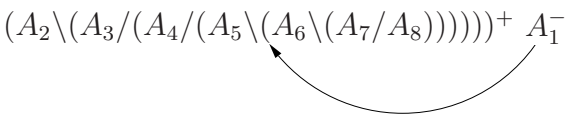
Given a proof frame $F = \langle V, E_R, E_L, O\rangle$, we will use the following shorthand:

- $\mathcal{V}(F) = V$
- $\mathcal{E}_{\mathcal{R}}(F) = E_R$
- $\mathcal{E}_{\mathcal{L}}(F) = E_L$
- $\mathcal{O}(F) = O$

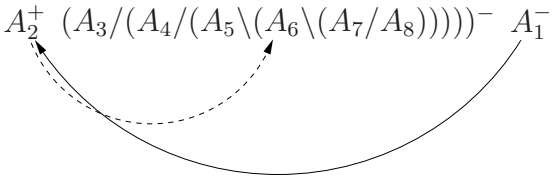
So far, we have defined proof frames for categories, but to develop an algorithm, we need to define proof frames for sequents, because the correctness results are proven based



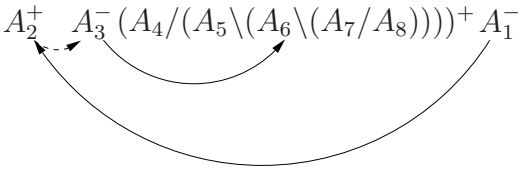
(a) The polarized category.



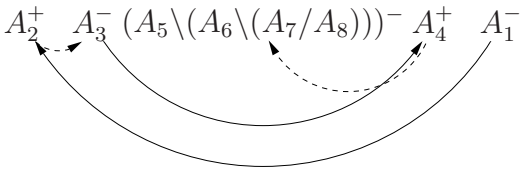
(b) After 1 vertex rewrite using rule (II).



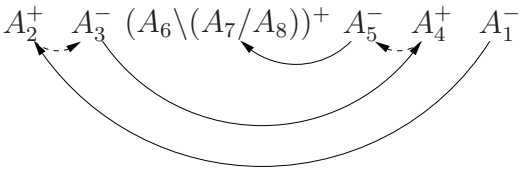
(c) After 2 vertex rewrites using rule (IV).



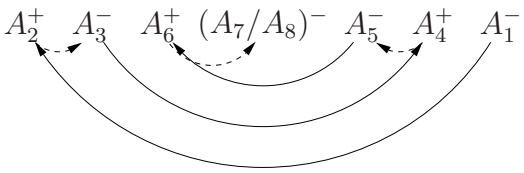
(d) After 3 vertex rewrites using rule (I).



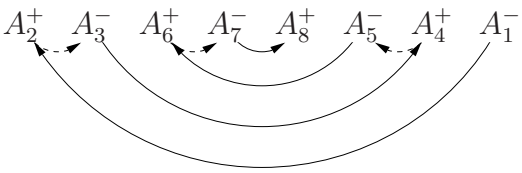
(e) After 4 vertex rewrites using rule (III).



(f) After 5 vertex rewrites using rule (II).



(g) After 6 vertex rewrites using rule (IV).



(h) The proof frame after using rule (I).

Figure 3.1: The derivation of the proof frame for the polarized category $(A_1\backslash(A_2\backslash(A_3/(A_4/(A_5\backslash(A_6\backslash(A_7/A_8)))))))^-$.

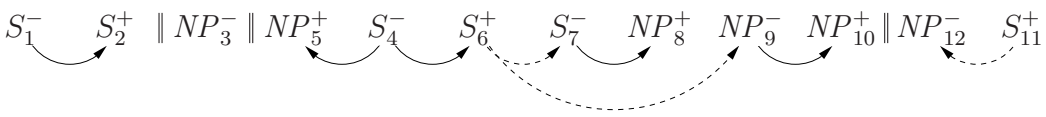


Figure 3.2: The proof frame for the sequent $S_1/S_2, NP_3, S_4\backslash NP_5/(S_6\backslash(S_7/NP_8)\backslash(NP_9/NP_{10})) \vdash S_{11}/NP_{12}$. The vertical bars denote the boundaries between atoms generated from different categories.

on sequents, and for sentence-lexicon pairs $\langle S, X\rangle$, since the input to the parsing problem is a sentence-lexicon pair.

Let $Q = c_1, \ldots, c_m \vdash c$ be a sequent. Let F_i be the proof frame for c_i^- for $1 \leq i \leq m$, let F be the proof frame for c^+ and let $\mathcal{F} = \{F_1, \ldots, F_m, F\}$. Then, the proof frame for Q is a quadruple $\langle V, E_R, E_L, O\rangle$ where:

- $V = \bigcup_{F\in\mathcal{F}} \mathcal{V}(F)$
- $E_R = \bigcup_{F\in\mathcal{F}} \mathcal{E}_{\mathcal{R}}(F)$
- $E_L = \bigcup_{F\in\mathcal{F}} \mathcal{E}_{\mathcal{L}}(F)$
- $O = \mathcal{O}(F_1) + \ldots + \mathcal{O}(F_m) + \mathcal{O}(F)$[4].

A proof frame for a sequent is shown in figure 3.2.

We will now extend the definition of proof frames to sentences, by defining a proof frame for a sentence-lexicon pair. Let $\langle S, X\rangle$ be a sentence-lexicon pair where $S = w_1, \ldots, w_l$. Let $w_{l+1} = \tau$, a new dummy word in the lexicon corresponding to the categories for a sentence. Then, each word w_i for $1 \leq i \leq l+1$ is assigned a set of categories by the lexicon. Let C_i be that set of categories and let C_i' be a sequence that is an arbitrary ordering of the category occurrences in C_i for $1 \leq i \leq l+1$. Then, let $F_{i,j}$

[4]We use + to denote concatenation of total orders where in the resultant total order, all elements in the set of atom occurrences of the left operand are less than all elements in the set of atom occurrences the right operand.

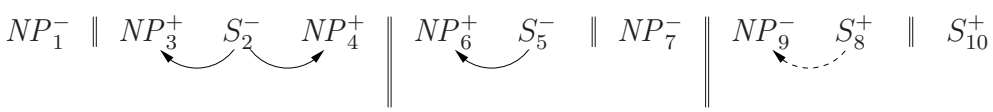


Figure 3.3: The proof frame for the sentence S = "Time flies" and the lexicon L = {Time: NP, Time: $S\backslash NP/NP$, flies: $S\backslash NP$, flies: NP, τ: $S\backslash NP$, τ: S}. Small vertical bars denote the boundaries between atoms belonging to categories assigned to the same word. Large vertical bars denote the boundaries between atoms for different words.

be the proof frame for the polarized category occurrence c^p where c is the j^{th} category occurrence in C_i' and p is negative for $1 \leq i \leq l$ and positive for $i = l + 1$. Finally, the proof frame for $\langle S, X \rangle$ is a quadruple $\langle V, E_R, E_L, O \rangle$ where:

- $V = \bigcup_{i,j} \mathcal{V}(F_{i,j})$
- $E_R = \bigcup_{i,j} \mathcal{E}_{\mathcal{R}}(F_{i,j})$
- $E_L = \bigcup_{i,j} \mathcal{E}_{\mathcal{L}}(F_{i,j})$
- $O = \left[\Sigma_j \mathcal{O}(F_{1,j})\right] + \ldots + \left[\Sigma_j \mathcal{O}(F_{l+1,j})\right]$.

A proof frame for a sentence-lexicon pair is shown in figure 3.3.

The intuition behind these definitions of proof frames is that the proof frames for sequents and sentence-lexicon pairs are defined in terms of proof frames for their categories. Categories corresponding to words receive negative polarity and categories corresponding to sentences receive positive polarity. Next, the vertex sets, and both edge sets of the proof frames for sequents and sentence-lexicon pairs are simply the union of the corresponding sets for the categories. The ordering is the concatenation of the orderings of the categories, only making sure that the ordering respects the ordering of categories in the sequent and words in the sentence.

When we discuss proof frames (among other structures) in later sections, we will refer to the sets of regular edges and Lambek edges using the usual graph-theoretic terms with

regular or Lambek prepended, respectively. For example, we might say that a graph is regular acyclic meaning that there is no cycle among the regular edges in a proof frame. We will also sometimes be interested in the graph consisting of a combination of both the regular and Lambek edges. In this case, we will simply use the usual graph-theoretic terms. For example, a graph is acyclic if there is no cycle consisting of regular and Lambek edges.

The following theorem begins to give us a sense of the kinds of structures that we will see in proof frames.

Proposition 3.1.1. *The in-degree of a vertex in a proof frame is either* 0 *or* 1*.*

Proof. By structural induction on the vertex rewriting rules. □

3.1.1 Abstract Proof Frames

Proof frames are a good representation of the structure of categories, but they represent too much information for use in an efficient algorithm. One of the issues is that the out-degree of a vertex in a proof frame is unbounded; therefore, we must find a way to succinctly represent the out-neighbourhoods of these vertices. Abstract proof frames are a representation of the same information in a proof frame, but represented in a more concise way than proof frames themselves. Abstract proof frames will provide the foundation for abstract term graphs, the structures that will be used in chart parsing. To proceed with our definition of abstract proof frames, we need to introduce the notion of the *sibling nodes* of a proof frame.

Definition 3.1.2. Let F be a proof frame. Let $A \subseteq \mathcal{V}(F)$ be a set of atom occurrences that share an in-neighbour in $\mathcal{V}(F)$. Then, for each such maximal set A, we define a sibling node b. For each A, we denote its sibling node b with the notation $\mathcal{B}(a)$, for any of the $a \in A$, or simply, $\mathcal{B}(A)$. Given a sibling node b, we denote A as $\mathcal{B}^{-1}(b)$ and we call

the atoms in $\mathcal{B}^{-1}(b)$ its *component atoms*. Then, the set of sibling nodes of F is the set of all such sibling nodes b.

Sibling nodes save us space in our representation because we can store information about the atom occurrences that share an in-neighbour in $\mathcal{V}(F)$ by referencing the sibling node rather than each component atom.

Abstract proof frames are a precise representation of proof frames, but require less space to store by using sibling nodes to represent redundant information. In particular, sibling nodes have only in-edges in an abstract proof frame, and atom occurrences have only out-edges.

Definition 3.1.3. Given a proof frame F, its *abstract proof frame* $APF(F)$ is a quadruple $\langle V, B, \hat{E_R}, \hat{E_L}\rangle$ where $V = \mathcal{V}(F)$, B is the set of sibling nodes of F, $\hat{E_R}$ is a set of regular edges and $\hat{E_L}$ is a set of Lambek edges which are defined as follows:

1. For each $\langle s, t\rangle \in \mathcal{E}_{\mathcal{R}}(F)$, $\langle s, \mathcal{B}(t)\rangle \in \hat{E_R}$
2. For each $\langle s, t\rangle \in \mathcal{E}_{\mathcal{L}}(F)$, $\langle s, \mathcal{B}(t)\rangle \in \hat{E_L}$

Given a proof frame F with abstract proof frame $\hat{F} = \langle V, B, \hat{E_R}, \hat{E_L}\rangle$, we will use the shorthand $\mathcal{V}(\hat{F}) = V \cup B$, $\mathcal{E}_{\mathcal{R}}(\hat{F}) = \hat{E_R}$ and $\mathcal{E}_{\mathcal{L}}(\hat{F}) = \hat{E_L}$. Based on this definition, we can trivially prove the following result concerning vertex degree for abstract proof frames:

Proposition 3.1.4. *Given a proof frame F and its abstract proof frame $\hat{F}$, sibling nodes in $\hat{F}$ have in-degree 1 and out-degree 0, and atom occurrences in $\hat{F}$ have in-degree 0 and out-degree 0 or 1.*

The abstract proof frame for the polarized category $(S_1\backslash NP_2/(S_3\backslash(S_4/NP_5)\backslash(NP_6/NP_7)))^-$ is shown in figure 3.4. We represent sibling nodes by $\mathcal{B}$ and with subscripts indicating their component atoms and for any component atoms appearing in the graph,

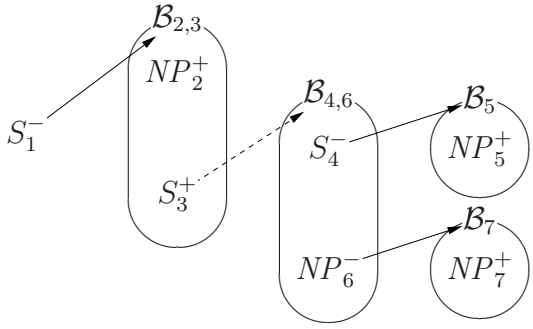


Figure 3.4: The abstract proof frame for the polarized category $(S_1\backslash NP_2/(S_3\backslash(S_4/NP_5)\backslash(NP_6/NP_7)))^-$.

we show them inside an oval with the sibling node at the top. Note that the total order is not represented in that figure. Also, note that polarities govern the types of edges, so the in-edges of siblings always have the same type.

3.1.2 Splits and Spans of Proof Frames

Our goal in developing proof frames was to better represent categories, but ultimately we will want to know how the categories in a sentence relate to each other. We will build up such a representation incrementally and, as such, we will need some notions regarding pieces of proof frames.

We will define the notion of a *split*, which will be a way of identifying the boundary between a piece of a proof frame that has already been analyzed and a piece of a proof frame that has yet to be analyzed. A *split* of a pair $\langle V, O\rangle$, where V is a vertex set and O is a total order, is a partition of the vertices in V into two sets such that the vertices in one half of the partition are all less than the vertices in the other half, according to O. The two sets of a split are the *left half* and *right half* of the split, where the left half is the half whose vertices are less than those in the right half. A split of a proof frame F is a split of $\langle \mathcal{V}(F), \mathcal{V}(O)\rangle$.

One way that we will reference splits is by the atom occurrences surrounding them. The *left split* of an atom occurrence a is the split whose left half consists of all those

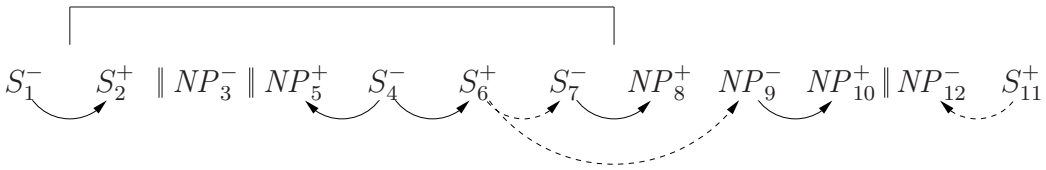


Figure 3.5: The span $[S_2, S_7]$ of the proof frame in figure 3.2

atoms that are less than a according to O and whose right half consists of a together with all those atoms that are greater than a. The *right split* of an atom occurrence a is the split whose left half consists of a together with all those atoms which precede a and whose right half consists of all those atoms which follow a. The intuition is that the left and right splits of an atom are the splits immediately adjacent to that atom.

Given our definitions of splits, we will now define a *span* to be a set of vertices in a proof frame that are contiguous[5] according to the total order. Given a pair of distinct splits of a proof frame, there is a unique set of atom occurrences appearing in the right half of one split and the left half of the other split. We call this the *middle* of the pair of splits. We can define a *span* simply as a pair of splits. We say that an atom occurrence is in a span if it appears in the middle of the pair of splits for the span. We denote a span by $[a_l, a_r]$ where a_l is the least atom occurrence in the span and a_r is the greatest atom occurrence in the span, according to O. Of the two splits for the span, the *left split* is the split whose right half contains a_l and the *right split* is the split whose left half contains a_l. We extend the notion of an atom occurrence being in a span to sibling nodes as follows: a sibling node is in a span if all of its component atom occurrences are in the span.

Figure 3.5 shows the proof frame from figure 3.2 with the span $[S_2, S_7]$ depicted above the proof frame. The atom occurrences in that span are S_2, NP_3, NP_5, S_4, S_6 and S_7. Its left split is the split whose left half is the singleton set consisting of S_1 and whose right half is the set $S_2, NP_3, \ldots, NP_{12}, S_{11}$. Its right split is the split whose left half is

the set $\{S_1, S_2, \ldots, S_6, S_7\}$ and whose right half is the set $NP_8, NP_9, NP_{10}, NP_{12}, S_{11}$.

We will now begin to define a relationship between edges in abstract proof frames and the splits of a proof frame. We say that an edge $\langle s, T\rangle$ of an abstract proof frame *crosses* a split if s is in one half of the split and at least one component atom of T is in the other half.

Our next theorem references paths in an abstract proof frame, and as such, we need a more formal definition of paths in both proof frames and abstract proof frames. We are perhaps more formal than necessary here because in section 3.4, we will define an additional notion of paths.

Definition 3.1.5. A *path* P through a set of vertices V and a set of edges E is a sequence of vertices $\langle v_1, \ldots, v_l\rangle$ such that $v_i \in V$ for $1 \leq i \leq l$ and the edge $\langle v_i, v_{i+1}\rangle \in E$ for $1 \leq i < 1$. We will refer to v_1 as the *initial vertex* of P, v_l as the *final vertex* of P and the vertices $v_2, \ldots, v_{l-1}$ as the *intermediate vertices* of P. We will refer to the initial vertex and final vertex as the *endpoints* of a path.

Let F be a proof frame or an abstract proof frame. Then, the *set of regular paths* of F, $\mathcal{P}_{\mathcal{R}}(F)$, is the set of paths over $\mathcal{V}(F)$ and $\mathcal{E}_{\mathcal{R}}(F)$. Furthermore, the *set of paths* of F, $\mathcal{P}(F)$, is the set of paths over $\mathcal{V}(F)$ and $\mathcal{E}_{\mathcal{R}}(F) \cup \mathcal{E}_{\mathcal{L}}(F)$.

Note that our definition of paths allows for empty paths, i.e., paths consisting of a single vertex. We can now specify an important proposition that states the types of structures that can be found crossing a split in a proof frame.

Proposition 3.1.6. *Given a proof frame F and its abstract proof frame $\hat{F}$ and a split T of F, let E be the set of edges in $\mathcal{E}_{\mathcal{R}}(\hat{F}) \cup \mathcal{E}_{\mathcal{L}}(\hat{F})$ that cross T. Then, there exists a path P in F whose atom occurrences all belong to a single category such that the sources of the edges in E are all on the path P.*

[5]Contiguity of a subset according to a total order is a property that specifies that every element not in the subset is either less than or greater than all elements in the subset.

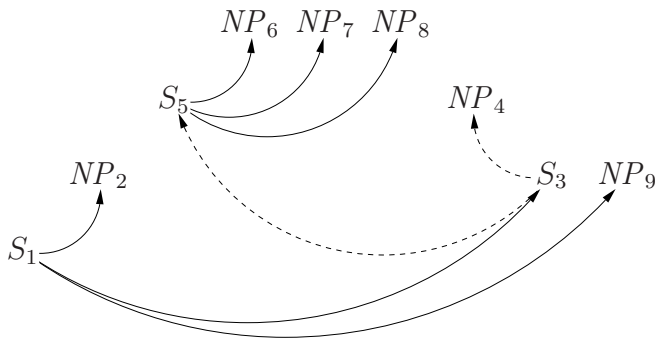


Figure 3.6: The proof frame for the polarized category $(S_1/NP_2/(S_3\backslash NP_4\backslash(S_5/NP_6/NP_7/NP_8))/NP_9)^-$, depicted to emphasize the proof frame as a tree.

Proof. By structural induction on the proof frame decomposition rules. □

Proposition 3.1.6 will be the key to the proofs that chart parsing can be done in polynomial-time. In informal terms, it states that for a given split there is a spine in the proof frame along which the split separates the proof frame into two halves. Figure 3.6 demonstrates this by showing the proof frame for the category emphasized as a tree, which illustrates that for any given split, only edges incident to the spine cross it. Note that wherever the split appears, a maximum of 3 vertices have out-edges crossing it, which is the order of the category.

Finally, we will need definitions regarding the vertices in a proof frame and in an abstract proof frame that are near a split. These vertices will be important as they are the only vertices necessary to represent the partial states of our algorithm.

Definition 3.1.7. An atom or sibling node is *immediate peripheral* to a span $[a_l, a_r]$ if it is not in $[a_l, a_r]$ and it is incident to an edge in the abstract proof frame that crosses $[a_l, a_r]$.

Definition 3.1.8. An atom is *peripheral*[6] to a span if it is immediate peripheral to the

[6]We need to include the ancestors of immediate peripheral vertices in the definition of peripheral because in the Adjoining algorithm, we must consider the possibility of adjoining two ATGs that have

span or if it is an ancestor in the proof frame of an immediate peripheral atom and it is not in the span. A sibling node is *peripheral* to a span if it is immediate peripheral to the span or if one of its component atoms is an ancestor in the proof frame of an immediate peripheral atom and it is not in the span.

The intuition behind the definition of immediate peripheral is that the vertices that are incident to edges that cross the span are immediate peripheral. The intuition behind peripheral is that all ancestors of immediate peripheral vertices and all immediate peripheral vertices are peripheral.

In figure 3.5, S_1 is an atom that is immediate peripheral to the span, due to its proximity to the left split. As for vertices peripheral to the span due to their proximity to the right split, the sibling nodes $\mathcal{B}(NP_8)$ and $\mathcal{B}(S_7, NP_9)$ are immediate peripheral. There are no atoms or sibling nodes that are peripheral to the span that are not immediate peripheral, because such peripheral vertices only appear in categories with very long paths where the span appears entirely within the category.

3.1.3 Pre-calculation of the Abstract Proof Frame

Abstract proof frames give us a way of representing the immediate children of vertices in a proof frame without having to reference those vertices directly. Without sibling nodes, in the worst case, we would need to iterate over all of the children of a vertex at various points in the algorithm. To be able to use abstract proof frames in our algorithm, we need access to some information that must be calculated before we begin chart parsing.

The information that is required is the edges in an abstract proof frame that cross a given split. Let F be a proof frame, let $\hat{F}$ be its abstract proof frame and let T be a split of that frame. Then, we define $CE_F(T)$ to be the set of edges in $\hat{F}$ that cross T.

a path between them through peripheral, but not immediate peripheral, vertices. To establish the representativeness of the resulting ATG, we need the path through those vertices, so they must be included in the ATG.

Furthermore, we define $CEA_F(T)$ to be $CE_F(T)$, along with any edge in $\hat{F}$ whose target is an ancestor in $\hat{F}$ of an endpoint of an edge in $CE_F(T)$. We can calculate these sets by simply iterating over each edge in the proof frame, and comparing the atom that is its source and the leftmost and rightmost atoms in the component atoms of the sibling node that is its target and comparing them to the split. The calculation for the entire proof frame can be done in time $O(n^2)$, relative to the number of atoms in the proof frame, since there are a linear number of edges in the abstract proof frame.

3.2 Term Graphs

The LC-Graphs of Penn (2004) are functionally sufficient to provide the representation of LCG parses in an efficient parsing algorithm, but they are needlessly complex in requiring an intermediate representation of variables, and also in both the number of correctness conditions and the precise wordings of their definitions. In this section, we will introduce *term graphs*[7], a proof net representation closely related to, but simpler than, Penn's LC-Graphs.

Proof nets are a graphical representation of the structure of the categories for the words in a sentence and the connections between those categories. Section 3.1 introduced the deterministic portion of proof nets that we will be using, and in this section we will introduce the non-deterministic portion that completes the definition of term graphs. We have separated the definitions in this way because proof frames will be the foundation of additional mathematical structure in sections 3.3 and 3.4. Term graphs are defined only for sequents (as opposed to proof frames which we defined for categories, sequents and sentence-lexicon pairs) because sequents are the structures over which LCG parses are defined.

Two other authors have developed variants of proof nets that are quite similar to

[7]We call them term graphs because they are a graphical representation of the semantic term.

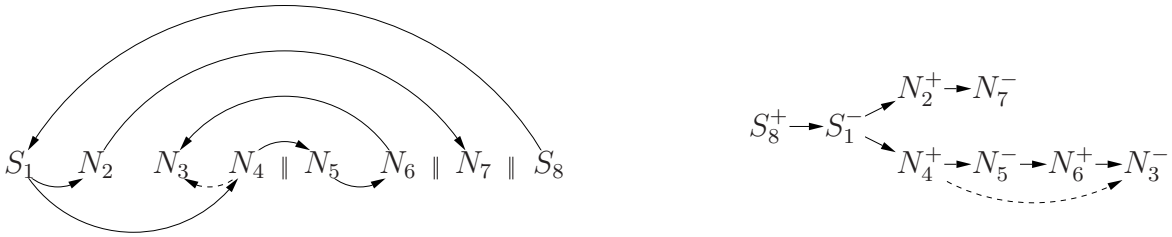
the term graphs defined here and were developed independent of our work. François Lamarche circulated an unpublished manuscript that is commonly cited in the literature as an Imperial College Technical Report in either 1994 or 1995. However, no such technical report exists at Imperial College. That unpublished manuscript was eventually published as Lamarche (2008). Lamarche's essential nets are proof structures for linear logic, but share much of the same structure and correctness conditions as term graphs. Concurrent to our research on term graphs, Savateev (2008) developed an algebra that is used to prove that the sequent derivability problem for the Lambek Calculus without product is NP-complete. Savateev's algebra is structurally quite similar to term graphs insofar as there is a mapping between the structures of the two systems and the correctness conditions of the two systems (Fowler, 2009).

We will now introduce some basic definitions for term graphs. Let $Q = c_1, \ldots, c_m \vdash c$ be a sequent and let F be its proof frame.

Definition 3.2.1. A *linkage* K for a proof frame F is a set of pairs of vertices in $\mathcal{V}(F)$ such that each vertex in $\mathcal{V}(F)$ appears exactly once among the sets of pairs, and each pair contains one positive vertex and one negative vertex. Given a linkage K for a proof frame F, we say that there is a *link* between vertices in each pair in K, and given a vertex $v \in \mathcal{V}(F)$, we denote the vertex to which it is linked as $\mathcal{K}_K(v)$. $\mathcal{K}_K$ is symmetric, i.e. if $K_K(a) = b$, then $K_K(b) = a$.

Definition 3.2.2. A *term graph* G for a sequent Q is a pair $\langle F, K \rangle$ where F is a proof frame for Q and K is a linkage for F. The vertex set $\mathcal{V}(G)$, the set of Lambek edges $\mathcal{E}_{\mathcal{L}}(G)$ and the total order $\mathcal{O}(G)$ of G are the same as the vertex set, the set of Lambek edges and the total order of F, respectively. However, the regular edge set $\mathcal{E}_{\mathcal{R}}(G)$ of G is the set of regular edges of the proof frame together with a set of edges for each pair in K consisting of a regular edge from the positive vertex to the negative vertex in the pair.

Due to its non-commutative nature, computing all of the parses for LCG requires only considering those linkages that are planar in the half plane, or rather, those linkages that



(a) The term graph depicted according to the total order.

(b) The term graph depicted with the root on the left and the leaves on the right.

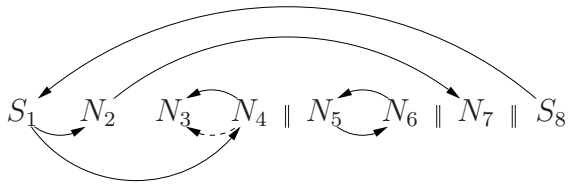
Figure 3.7: Two depictions of an integral term graph for the sequent $(S/N)/(N/N), N/N, N \vdash S$. We will no longer depict polarities because they can be inferred from lemma 3.2.3.

can be drawn without crossing edges above the proof frame. The extent to which term graphs are non-deterministic is that for a given sequent there are a possibly exponential number of linkages that correspond to a possibly exponential number of term graphs. Perhaps the most important property of our definition of proof nets is that we will be able to prove that they are sufficient for representing proofs in LCG.

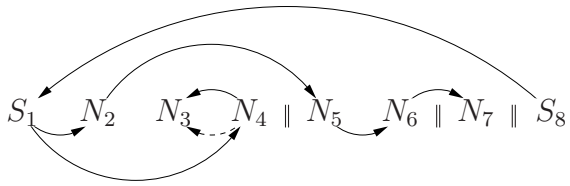
Figure 3.7 shows examples of term graphs. When we depict term graphs with the linkage edges shown above the vertices and the proof frame edges shown below, as in figure 3.7a, we can use the linkage edges to determine the polarity of the vertices so polarities will sometimes not be specified explicitly. Otherwise, we will specify the polarities, as in figure 3.7b.

We will now proceed with some preliminaries that will help us to begin to understand the structure of term graphs, before we provide our correctness conditions.

Lemma 3.2.3. *The vertices in a term graph have the following restrictions on degree:*



(a) A term graph that is not L^*-integral.



(b) A term graph that is L^*-integral but not integral.

Figure 3.8: Term graphs for the sequent $(S/N)/(N/N), N/N, N \vdash S$.

	Negative vertices	*Positive vertices*
regular in-degree	1	1
regular out-degree	*Unbounded*	1
Lambek in-degree	≤ 1	0
Lambek out-degree	0	*Unbounded*

Proof. By structural induction on the vertex rewriting rules. □

We can now define the correctness conditions for term graphs.

Definition 3.2.4. Given a term graph G, we define the following three conditions on it:

T(1): G is regular acyclic.

T(2): For all Lambek edges $\langle s, t\rangle$ in G, there is a regular path from s to t.

T(CT): For each Lambek edge $\langle s, t\rangle$, there exists a negative vertex x such that there is a regular path from s to x and there is no Lambek edge $\langle s', x\rangle$ such that there is a regular path from s to s'.

A term graph is L^*-*integral* iff it satisfies $\mathbf{T}(1)$ and $\mathbf{T}(2)$. It is *integral* iff it is L^*-*integral* and satisfies $\mathbf{T}(\mathrm{CT})$.

3.2.1 Correctness

In definition 3.2.4, we provided conditions on a term graph that allowed us to define two types of integrity on term graphs: one for L, the Lambek calculus banning empty premises, and for L^*, the Lambek calculus allowing empty premises. These definitions of integrity correspond to the existence of proofs in L and L^*, respectively, and we must now prove this. We will not be proving that our integrity conditions correspond to proofs in L and L^* directly, but rather via the LC-Graphs of Penn (2004), which we introduced in section 2.5.3.

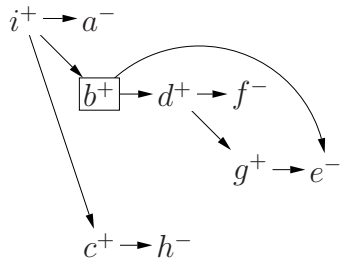
To review briefly, LC-Graphs are constructed using decomposition rules that assign terms to category occurrences in a sequent and produce side conditions. Then, an LC-Graph G is a graph whose vertex set $\mathcal{V}(G)$ is the set of variables that are assigned to some atom occurrence in the sequent or appear in a side condition. Edges in an LC-Graph are specified according to which variables occur in which terms and according to the variables occurring in the side conditions generated during the counterpart to vertex rewriting. Finally, the integrity conditions for LC-Graphs are defined as conditions on paths in LC-Graphs. Section 2.5.3 has the formal definition. We will be proving the correctness of the integrity conditions by showing a correspondence between the vertices and edges in an LC-Graph for a linkage and the vertices and edges for the term graph for the structurally isomorphic linkage. Then, we will prove that the integrity conditions are equivalent. Figure 3.9 depicts the application of the decomposition rules and the resulting LC-Graph for the sequent from figure 3.7.

First, we must repeat a definition from Penn (2004) which is necessary before we proceed:

Definition 3.2.5. A *lambda-node* is a positive vertex in an LC-Graph that appears as

$$
\begin{aligned}
& ((S_1/N_2)/(N_4/N_3))^- : a, (N_5/N_6)^- : f, N_7^- : h, S_8^+ : i \\
\rightarrow\ & (S_1/N_2)^- : ab, (N_4/N_3)^+ : b, (N_5/N_6)^- : f, N_7^- : h, S_8^+ : i \\
\rightarrow\ & S_1^- : abc, N_2^+ : c, (N_4/N_3)^+ : b, (N_5/N_6)^- : f, N_7^- : h, S_8^+ : i \\
\rightarrow\ & S_1^- : abc, N_2^+ : c, N_3^- : e, N_4^+ : d, (N_5/N_6)^- : f, N_7^- : h, S_8^+ : i \qquad [b := \lambda e.d] \\
\rightarrow\ & S_1^- : abc, N_2^+ : c, N_3^- : e, N_4^+ : d, N_5^- : fg, N_6^+ : g, N_7^- : h, S_8^+ : i \qquad [b := \lambda e.d]
\end{aligned}
$$

(a) The decomposition of the sequent.



(b) The LC-Graph.

Figure 3.9: The decomposition and the LC-Graph corresponding to the term graph shown in figure 3.7. Lambda-nodes are indicated with a box around the vertex.

the left side of a side condition.

Then, because of the definition of LC-Graphs, a lambda-node has exactly one negative out-neighbour and one positive out-neighbour. In figure 3.9, b is a lambda-node. The intuition is that the edge from a lambda-node to its negative out-neighbour corresponds to a Lambek edge in a term graph. This relationship will be made formal in definition 3.2.7.

We will prove the equivalence of term graphs and LC-Graphs via a graph transformation from LC-Graphs to term graphs. To do this, we require a definition that helps

us overcome the lack of the representation of lambda-nodes in term graphs.

Definition 3.2.6. A *lambda-node chain* $C = \{c_1, \ldots, c_m\}$ is a maximal path of lambda-nodes in an LC-Graph. The *endpoint* of a lambda-node chain is the positive out-neighbour of c_m.

Definition 3.2.7. Given an LC-Graph H, we define the graph transformation Θ from LC-Graphs to term graphs on vertices:

- For non-lambda-nodes $v \in \mathcal{V}(H)$, $\Theta(v)$ is the atom occurrence a for which v is the leftmost variable in the term label of a assigned by the proof frame of H
- For lambda-nodes $v \in \mathcal{V}(H)$, $\Theta(v)$ is $\Theta(e)$ where e is the endpoint of the lambda-node chain to which v belongs

and edges:

- For an edge $\langle s, t\rangle \in \mathcal{E}(H)$, where s is positive and not a lambda-node and t is negative, $\Theta(\langle s, t\rangle)$ is the regular edge $\langle \Theta(s), \Theta(t)\rangle$
- For an edge $\langle s, t\rangle \in \mathcal{E}(H)$, where s is positive and not a lambda-node and t is positive, $\Theta(\langle s, t\rangle)$ is the regular edge $\langle \Theta(o), \Theta(t)\rangle$ where o is the negative out-neighbour of s in H
- For an edge $\langle s, t\rangle \in \mathcal{E}(H)$ where s is a lambda-node and t is positive, $\Theta(\langle s, t\rangle)$ is null
- For an edge $\langle s, t\rangle \in \mathcal{E}(H)$, where s is a lambda-node and t is negative, $\Theta(\langle s, t\rangle)$ is the Lambek edge $\langle \Theta(e), \Theta(t)\rangle$ where e is the endpoint of the lambda-node chain to which s belongs

Then, $\Theta(H)$ is defined as applying Θ to the vertices and edges of H.

The intuition of Θ consists of the following two points:

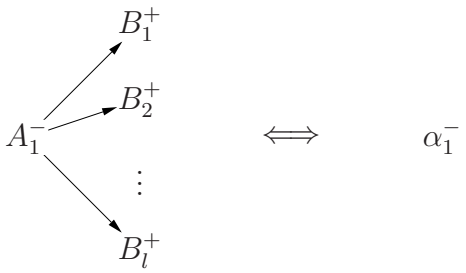


Figure 3.10: Θ applied to the out-neighbourhood of a negative vertex. The term graph is on the left and the LC-Graph is on the right. The variable α_1 corresponds to the atom occurrence A_1.

1. The vertices in a term graph are locally rearranged relative to the variables in an LC-Graph.

2. Term graphs do not contain lambda-nodes, but represent the same information using Lambek edges.

The first of these differences is required for term graphs to avoid the introduction of terms that are required by LC-Graphs. The second difference allows us to express our correctness conditions more concisely and simplifies the presentation of our proofs. This is accomplished by no longer needing to resort to those edges in an LC-Graph that are the equivalent of Lambek edges by referring to their endpoint vertices or the existence of lambda-nodes.

We should first notice that Θ transforms lambda-nodes to the same atom occurrences as the endpoints of their lambda-node chains and that edges from lambda-nodes to their positive out-neighbours are not transformed to anything. This means that Θ as defined on vertices and edges is not invertible. However, it is not difficult to prove that Θ, as defined on graphs, is invertible, modulo the choice of variables and the ordering of the vertices within a lambda-node chain. A depiction of Θ is shown in figures 3.10 and 3.11.

We can now define the correctness conditions on LC-Graphs and state theorem 3.2.8

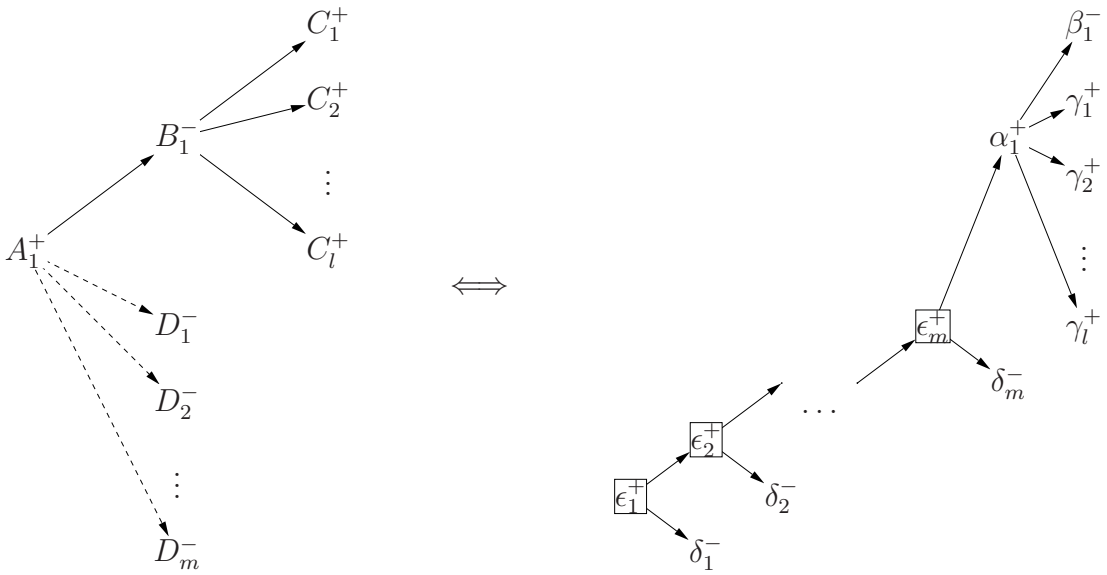


Figure 3.11: Θ applied to the out-neighbourhood of a positive vertex. The term graph is on the left and the LC-Graph is on the right. The Latin letters in the term graph correspond to their Greek equivalents in the LC-Graph. The positive vertices $\epsilon_1, \ldots, \epsilon_m$ form a lambda-node chain. The edges $\langle \epsilon_i, \delta_i \rangle$ are represented in the term graph as the Lambek edges $\langle A_1, D_i \rangle$.

(proven in Penn (2004)):

I(1): There is a unique node in H with in-degree 0, from which all other nodes are path-accessible.

I(2): H is acyclic.

I(3): For every lambda-node $v \in \mathcal{V}(H)$, there is a path from its positive out-neighbour u to its negative out-neighbour w.

I(CT): For every lambda-node $v \in \mathcal{V}(H)$, there is a path in H, from v to x where x has out-degree 0 and there is no lambda-node $v' \in \mathcal{V}(H)$ such that there is a path from v to v' and $\langle v', x \rangle \in \mathcal{E}(H)$.

Theorem 3.2.8. *A sequent is derivable in L^* iff it has an LC-Graph satisfying conditions **I**(1-3). A sequent is derivable in L iff it has an LC-Graph satisfying conditions **I**(1-3) and **I**(CT).*

For the remainder of this section, we require some definitions of the objects related to our propositions. Let Q be a sequent. Let H be an LC-Graph with linkage K for the sequent Q. Let F be the proof frame for Q. Then, let K' be the structurally isomorphic linkage for F consisting of links between $\Theta(a)$ and $\Theta(b)$ for each link between a and b in K. Let G be the term graph $\langle F, K' \rangle$. First, we prove a lemma for paths over positive vertices in LC-Graphs that we will use repeatedly in the remainder of this section.

Lemma 3.2.9. *Let u and v be non-lambda-node positive vertices in H. Then, there is a path from u to v in H if and only if there is a regular path from $\Theta(u)$ to $\Theta(v)$ in G.*

Proof. Let there be a path from u to v in H. Because the out-degree of negative vertices in LC-Graphs is 0, any path from u to v must consist of only positive vertices. We will consider the edges in this path according to two types of subpaths: (1) those consisting of a lambda-node chain, viz. the positive in-neighbour to the first vertex in that lambda-node chain s and the endpoint of that lambda-node chain t and (2) edges from a non-lambda-node w to a non-lambda-node x. For subpaths consisting of a lambda-node chain and the non-lambda-node positive vertices incident to it, Θ transforms this path to two regular edges, one from $\Theta(s)$ to an intermediate negative vertex and one from that intermediate negative vertex to $\Theta(t)$. For subpaths consisting of edges between non-lambda-nodes, Θ also transforms them into two regular edges, one from $\Theta(w)$ to an intermediate negative vertex and one from that negative vertex to $\Theta(x)$. In both cases, the regular path in G is established.

Let there be a regular path from $\Theta(u)$ to $\Theta(v)$ in G. First, note that regular paths in term graphs alternate between positive and negative vertices. Let v_i, v_{i+1} and v_{i+2} be three consecutive vertices in the path from $\Theta(u)$ to $\Theta(v)$ such that v_{i+1} is negative. If the

non-lambda-node vertex in H that Θ transforms to v_{i+2} is the endpoint of a lambda-node chain, then according to Θ, there must be an edge from the non-lambda-node vertex that Θ transforms to v_i to the initial vertex in that lambda-node chain. If the non-lambda-node vertex in H that Θ transforms to v_{i+2} is not the endpoint of a lambda-node chain, then, there is an edge from the non-lambda-node vertex that Θ transforms to v_i to the non-lambda-node vertex that Θ transforms to v_{i+2}. In both cases, the path is established in H. □

Next, we will prove that Θ applied to H yields the same term graph as building the term graph from F and K'.

Proposition 3.2.10. $G = \Theta(H)$[8]

Proof. First, consider a link between a and b in K where a is positive and b is negative. Such a link introduces edges into H from the variable labelling a to each variable appearing in the term labelling b. The linkage edge from $\Theta(a)$ to $\Theta(b)$ in K' only introduces an edge into G from $\Theta(a)$ to $\Theta(b)$. In the decomposition rules to build H, no edges were introduced between the variable labelling a and the variables appearing in the label of b, but in the vertex rewriting rules to build G, edges were introduced from $\Theta(b)$ to $\Theta(c)$ of each positive c appearing in the label of b.

Then, consider a term P labelling a positive non-atom category in the decomposition rules to build H. When this term is decomposed, a side condition $[x := \lambda z.y]$ is generated, which in turn generates edges $\langle x, z\rangle$ and $\langle x, y\rangle$ in $\mathcal{E}(H)$. Now, consider the point in the building of G where the category labelled by P is rewritten. The only edge that is introduced is a Lambek edge from the resulting positive category to the resulting negative category. After vertex rewriting is finished, that Lambek edge will be an edge from the innermost atom of the resulting positive category to the innermost atom of the resulting negative category. □

[8] We can state this as a true graph equivalency, since the vertices in each of the graphs originate from the underlying categories.

We will now prove that the **I** conditions are satisfied by H if and only if the **T** conditions are satisfied by G.

Proposition 3.2.11. ***I****(2)* $\Leftrightarrow$ ***T****(1)*

Proof. Suppose there is a cycle $C = \langle c_1, \ldots, c_m \rangle$ in H where $c_m = c_1$. Since the out-degree of negative vertices in an LC-Graph is always 0, any cycle in an LC-Graph must contain only positive vertices. Then, according to lemma 3.2.9, there is a regular cycle in G.

Now, suppose there is a regular cycle $C = \langle c_1, \ldots, c_m \rangle$ in G where $c_m = c_1$ and, without loss of generality, let it begin and end with a positive vertex. Let v be a vertex in H such that $\Theta(v) = c_1$. Then, according to lemma 3.2.9, there is a cycle in H. □

Proposition 3.2.12. ***I****(3)* $\Leftrightarrow$ ***T****(2)*

Proof. Let s be a lambda-node in $\mathcal{V}(H)$, let t be its positive out-neighbour and u be its negative out-neighbour. We will consider any paths from t to u in three pieces: (1) the portion from t to the endpoint e of the lambda-node chain of s, (2) the portion through other positive vertices and (3) the final edge in that path from a positive vertex p to u.

According to Θ, the edge $\langle s, u \rangle \in \mathcal{E}(H)$ is transformed to a Lambek edge from $\Theta(e)$ to $\Theta(u)$. Then, there is a path in H from e to p if and only if there is a path from $\Theta(e)$ to $\Theta(p)$, by lemma 3.2.9. In addition, the edge from p to u is transformed into a regular edge from $\Theta(p)$ to $\Theta(u)$ in G.

Then, there is a path from t to u in H if and only if there is a regular path from $\Theta(e)$ to $\Theta(u)$ in G and there is a Lambek edge from $\Theta(e)$ to $\Theta(u)$ in G. Therefore, $\mathbf{I}(3) \Leftrightarrow \mathbf{T}(2)$. □

Before we can proceed with proving an equivalence between $\mathbf{I}(1)$ and $\mathbf{I}(2)$, we must prove a pair of lemmas concerning vertex in-degree in LC-Graphs.

Lemma 3.2.13. *The in-degree of positive vertices in LC-Graphs is* 1 *except for a single positive vertex* t *that has in-degree* 0*.*

Proof. From section 2.5.3, we can see that positive vertices in an LC-Graph are introduced either in a decomposition rule applied to a negative category (in rules (I) or (II)), in a decomposition rule applied to a positive category (in rules (III) and (IV)) or they are the principal functor of the output category.

For positive vertices of the first type, such vertices appear in the label of exactly one negative atom, and the axiomatic linkage introduces exactly one in-edge from this. For positive vertices of the second type, the in-degree is 1 because the edges introduced due to the side condition are the only in-edges. For the positive vertex of the third type, the in-degree is 0, because neither the axiomatic linkage nor any of the four decomposition rules introduce an in-edge to it. □

Lemma 3.2.14. *The out-degree of negative vertices in LC-Graphs is* 0*.*

Proof. Neither the axiomatic linkage, nor any of the decomposition rules introduce any out-edges to negative vertices. □

Then, we can proceed with a proof of the equivalence between $\mathbf{I}(1)$ and $\mathbf{I}(2)$.

Proposition 3.2.15. ***I****(1)* $\Leftrightarrow$ ***I****(2)*

Proof. First, suppose that H contains a cycle. By lemma 3.2.14, any cycle must consist of only positive vertices. Then, by lemma 3.2.13, $\mathbf{I}(1)$ can only be satisfied if all positive vertices in H are on that cycle. However, this is impossible since there is a positive vertex with in-degree 0.

Now, suppose that H does not contain a cycle. By lemma 3.2.13, there is exactly one positive vertex with in-degree 0. Let t be that vertex. Then, since t has in-degree 0, all other positive vertices have in-degree 1 and there are no cycles, for any given positive vertex p, we can determine the path from t to p in reverse by repeatedly adding the edge from a vertex to its in-neighbour. Furthermore, negative vertices in LC-Graphs all have a positive in-neighbour. Therefore, there must be a path from t to every vertex in H. □

Proposition 3.2.16. ***T****(CT)* $\Leftrightarrow$ ***I****(CT)*

Proof. Suppose that for every lambda-node $v \in \mathcal{V}(H)$, there is a path P from v to some vertex x such that x has out-degree 0 and there is no lambda-node $v' \in \mathcal{V}(H)$ such that there is a path from v to v' and $\langle v', x\rangle \in \mathcal{E}(H)$. Let u be the negative out-neighbour of v in H. Then, Θ transforms the edge $\langle v, u\rangle \in \mathcal{E}(H)$ to the Lambek edge $\langle \Theta(v), \Theta(u)\rangle \in \mathcal{E}_{\mathcal{L}}(G)$. Since negative vertices have out-degree 0 in LC-Graphs, we know that the path from v to the penultimate vertex p in P is made up of positive vertices. By lemma 3.2.9, we know that there is a regular path from $\Theta(v)$ to $\Theta(p)$. In addition, Θ transforms edges from positive vertices to negative vertices in LC-Graphs to regular edges in term graphs, which establishes a regular path from $\Theta(v)$ to $\Theta(x)$. Then, if there was a Lambek edge $\langle s', \Theta(x)\rangle \in \mathcal{E}_{\mathcal{L}}(G)$ for some s' such that there was a regular path from $\Theta(v)$ to s', then according to Θ, there would exist a lambda-node $v' \in \mathcal{V}(H)$ such that $\Theta(v') = s'$ and $\langle v', x\rangle \in \mathcal{E}(H)$. By lemma 3.2.9, there is a path from v to v', which contradicts our assumption.

Now, suppose there is some lambda-node $v \in \mathcal{V}(H)$ such that for every vertex x with out-degree 0 for which there is a path from v to x, there is also a path from v to a lambda-node v' where $\langle v', x\rangle \in \mathcal{E}(H)$. Let u be the negative out-neighbour of v in H. Consider a path in G from $\Theta(v)$ to some negative vertex x. If there is not a Lambek edge $\langle t, x\rangle \in \mathcal{E}_{\mathcal{L}}(G)$ such that there is a regular path from $\Theta(v)$ to t, then, by lemma 3.2.9, there cannot be a lambda-node v' such that there is a path in H from v to v', or else t could be $\Theta(v')$ which contradicts our assumption. □

These results, together with theorem 3.2.8 prove the following:

Theorem 3.2.17. *A sequent is derivable in L iff it has a term graph satisfying* ***T****(1) and* ***T****(2). A sequent is derivable in* L^* *iff it has a term graph satisfying conditions* ***T****(1),* ***T****(2) and* ***T****(CT).*

3.3 Partial Term Graphs

Term graphs, introduced in section 3.2, give us a representation of proofs in Lambek Categorial Grammar (LCG) that is easier to deal with computationally than either the classical proofs for LCG or the proof nets of Penn (2004) introduced in section 2.5. Our goal with term graphs is to provide an efficient algorithm, which, similar to many of the other results in the parsing literature, takes the form of a chart parsing algorithm. Before we can use term graphs in a chart parsing algorithm, we will need the notion of a *partial term graph* (PTG). The intuition behind PTGs is that we use the same proof frames as were introduced for term graphs, but we now consider linkages that link only a subset of the atom occurrences in the proof frame.

Definition 3.3.1. A *partial linkage* K for a proof frame F is a set of pairs of vertices in $\mathcal{V}(F)$ such that a given vertex appears *at most once* among the set of pairs, and each pair contains one positive vertex and one negative vertex.

We will only consider *contiguous* partial linkages, or rather, partial linkages such that the vertices in the set of pairs of vertices are contiguous according to $\mathcal{O}(F)$.

Definition 3.3.2. A *partial term graph* (PTG) for a sequent Q is a pair $\langle F, K \rangle$, where F is the proof frame for Q and K is a contiguous partial linkage of F.

For partial term graphs, we will define the vertex set, the regular edge set, Lambek edge set and total order in the same way as we did for term graphs. And, as with term graphs, we will refer to the vertex set, regular edge set, Lambek edge set and total order of a PTG G as $\mathcal{V}(G)$, $\mathcal{E}_{\mathcal{R}}(G)$, $\mathcal{E}_{\mathcal{L}}(G)$ and $\mathcal{O}(G)$, respectively. Furthermore, we define the *span* of G, $\mathcal{N}(G)$, as the pair of splits consisting of the left split of the leftmost atom in K and the right split of the rightmost atom in K. This means that the middle of $\mathcal{N}(G)$ consists of exactly the atoms in K. We will specify the set of regular paths in a PTG G as $\mathcal{P}_{\mathcal{R}}(G)$. Figures 3.12 and 3.13 depict PTGs, without their spans explicitly marked, although their spans can be easily inferred from the linkage.

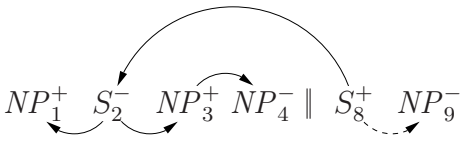


Figure 3.12: A PTG for the sequent $S_2\backslash NP_1/NP_3, NP_4 \vdash S_8\backslash NP_9$

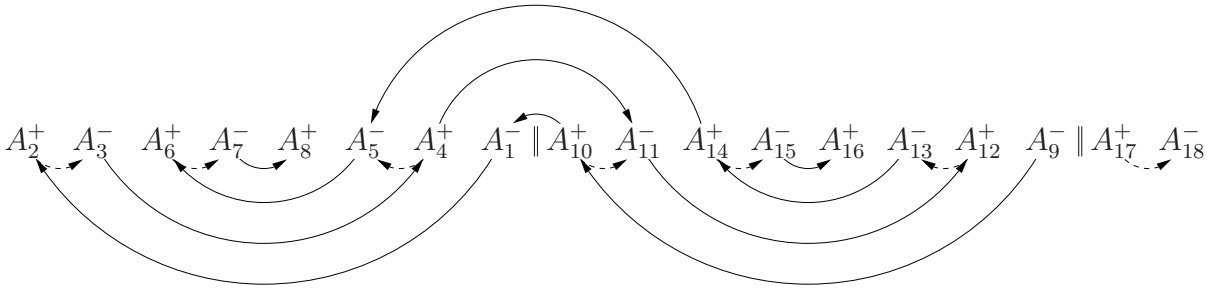


Figure 3.13: A PTG for the sequent $A_1\backslash(A_2\backslash(A_3/(A_4/(A_5\backslash(A_6\backslash(A_7/A_8))))))), A_9\backslash(A_{10}\backslash(A_{11}/(A_{12}/(A_{13}\backslash(A_{14}\backslash(A_{15}/A_{16}))))))) \vdash A_{17}\backslash A_{18}$

3.3.1 Incremental Integrity of Partial Term Graphs

The integrity conditions specified in section 3.2 gave us necessary and sufficient conditions for determining whether a term graph represents a valid proof in the Lambek Calculus. To achieve our goal of providing an algorithm for building term graphs, we will need some notion of whether a partial term graph might possibly be extensible to an integral term graph. To accomplish this, we will define a set of incremental integrity conditions. The intuition behind the incremental integrity conditions is that they specify whether there is some way of extending a partial term graph to become an integral term graph. First, we need the following definition concerning certain vertices in PTGs:

Definition 3.3.3. Given a PTG G and an atom $a \in \mathcal{V}(G)$, a is *open* iff a is positive and has no regular out-edge or a is negative and has no regular in-edge.

We should immediately notice that given a PTG G, an atom $a \in \mathcal{V}(G)$ is open iff a is not in the span of G, $\mathcal{N}(G)$. Note that PTGs are defined in terms of linkages, not

spans. This, combined with our preference for contiguous linkages, means that a PTG's span necessarily consists of closed vertices. Then, the incremental integrity conditions for PTGs are as follows:

Definition 3.3.4. Given a PTG G, we define the following three conditions on it:

$\mathbf{IT_P}(1)$: G is regular acyclic

$\mathbf{IT_P}(2)$: For each Lambek edge $\langle s, t\rangle$ in G, one of the following holds:

1. There is a regular path from s to t.
2. There is an open positive vertex u such that there is a regular path from s to u and there is an open negative vertex v such that there is a regular path from v to t.

$\mathbf{IT_P}(\mathrm{CT})$: For each Lambek edge $\langle s, t\rangle$ in G, one of the following holds:

1. There is a negative vertex x such that there is a regular path from s to x and there is no Lambek edge $\langle s', x\rangle$ such that there is a regular path from s to s'.
2. There is an open positive vertex u, an open negative vertex v and a negative vertex x such that there is a regular path from s to u, a regular path from v to x and there is no Lambek edge $\langle s', x\rangle$ such that there is a regular path from s to s' or from v to s'.

We say that a PTG that satisfies $\mathbf{IT_P}(1)$ and $\mathbf{IT_P}(2)$ is *incrementally L^*-integral*, and that a PTG that satisfies $\mathbf{IT_P}(1)$, $\mathbf{IT_P}(2)$ and $\mathbf{IT_P}(\mathrm{CT})$ is *incrementally integral*.

$\mathbf{IT_P}(1)$ is identical to $\mathbf{T}(1)$ because no regular acyclic term graph can have a subgraph with cycles. $\mathbf{IT_P}(2)$ and $\mathbf{IT_P}(\mathrm{CT})$ each have two clauses. The first clause in each is identical to their corresponding integrity conditions, which state that certain paths exist. The second clause in each covers the case where such a path does not yet exist. In both

$\mathbf{IT_P}(2)$ and $\mathbf{IT_P}(\mathrm{CT})$, the second clause specifies that the path is partially complete and that there are open vertices that can potentially be connected to complete the path.

Since these incremental integrity conditions are intended to specify which PTGs are extendible to integral term graphs, we must prove the following proposition.

Proposition 3.3.5. *A term graph is integral if and only if it is incrementally integral.*

Proof. A term graph does not have any open vertices. □

With this theorem, we will be able to assert that as long as we build all incrementally integral *partial* term graphs, then we will have built all integral term graphs which will give us the basis for our correctness proof.

3.4 Abstract Term Graphs

The introduction of term graphs gives us a simple method for determining whether a linkage is correct, and partial term graphs give us a method for determining whether pieces of a term graph could be extended to an integral term graph. However, partial term graphs contain too much information to be used in an efficient chart parsing algorithm. In this section, we introduce *abstract term graphs* (ATGs): an efficient representation of only the information in a partial term graph that is necessary to determine its integrity.

We will reduce the amount of information stored in ATGs, relative to PTGs, in three important ways. First, during chart parsing, regular paths in PTGs become long but the internal vertices on those paths are irrelevant for determining the integrity conditions. In ATGs, these paths are contracted. Second, vertices in PTGs share parts of their neighbourhood with their siblings in the proof frame (see definition 3.1.2). In ATGs, we have special vertices, called sibling nodes, for representing common neighbourhood information among sets of siblings. Third, during chart parsing, the only portion of the PTG that is relevant to determining the integrity conditions are those edges at the

border between the atoms with links and the atoms without links. In ATGs, vertices that are not peripheral to the span are not represented and when information concerning any vertices outside of the span of an ATG is needed, it is copied from the abstract proof frame.

In the preceding sections, we have provided integrity conditions on term graphs and partial term graphs for both L and L^*. However, because of the additional complexity of abstract term graphs, we will be providing definitions only for L^*. To accommodate L, the definition of ATGs would need to be modified slightly to contain additional information representing the path accessibility of certain vertices in the underlying PTG.

Definition 3.4.1. Given a proof frame F with abstract proof frame $\hat{F}$, an *abstract term graph (ATG)* for F is a quadruple $\hat{G} = \langle N, V, E_R, E_L \rangle$ where N is a span of F, V is a subset of the vertices in $\mathcal{V}(\hat{F})$, E_R is a set of directed regular edges between vertices in V and E_L is a set of directed hyperedges between vertices in V.

A directed hyperedge is an edge whose source is a set of vertices and whose target is a single vertex, and will be needed to represent sets of sibling nodes that are descendants of the sources of Lambek edges in the term graph. ATGs are abstract representations of PTGs, so they have similar restrictions on the types of edges and vertices that they may contain. There are the following restrictions on the neighbourhoods of vertices in ATGs:

1. Negative atoms have no Lambek out-edges
2. Positive atoms have no regular in-edges, no Lambek in-edges and their Lambek out-edges have sources that are singleton sets
3. Negative sibling nodes have no regular in-edges, no regular out-edges and no Lambek out-edges
4. Positive sibling nodes have no regular out-edges and no Lambek in-edges and their Lambek out-edges have sources that are sets containing only positive sibling nodes

Negative atoms cannot have Lambek out-edges, because Lambek out-edges indicate the source of a Lambek edge, which can always be traced to positive vertices. Positive atoms cannot have regular in-edges because any such edges are represented instead as in-edges of the positive sibling nodes of which they are a component atom. Positive atoms cannot have Lambek in-edges because Lambek in-edges indicate the target of a Lambek edge, which can always be traced to negative vertices. Positive atoms have Lambek out-edges whose sources are singleton sets because such Lambek edges represent the Lambek edge in the proof frame whose source is this positive atom. Negative sibling nodes cannot have any incident regular edges because negative sibling nodes are placeholders for the in-neighbourhood of their component atoms. Negative sibling nodes cannot have Lambek out-edges for the same reason that negative atoms cannot. Positive sibling nodes cannot have regular out-edges because any such edges are represented instead as out-edges of their component atoms. Positive sibling nodes cannot have Lambek in-edges for the same reason that positive atoms cannot. Finally, positive sibling nodes have Lambek out-edges whose sources are sets containing only positive sibling nodes because such Lambek edges represent a Lambek edge in the abstract proof frame whose source is not open.

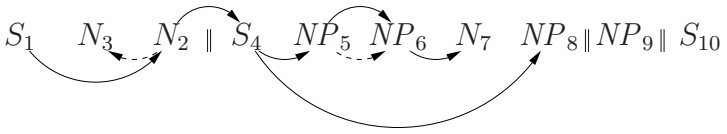
As with our previous definitions of graphs, for an ATG $\hat{G} = \langle N, V, E_R, E_L \rangle$, we will use $\mathcal{V}(\hat{G})$ to refer to the set V, $\mathcal{E}_{\mathcal{R}}(\hat{G})$ to refer to the set E_R and $\mathcal{E}_{\mathcal{L}}(\hat{G})$ to refer to the set E_L. In addition, we will use $\mathcal{N}(\hat{G})$ to refer to the span N.

Subfigure (c) of figure 3.14 and subfigures (b-e) in figures 3.15 and 3.16 depict ATGs. Sibling nodes are represented by $\mathcal{B}$ and a subscript indicating their component atoms. An oval is used to show any component atoms of a sibling node that appear in the ATG. Note that the neighbourhood of a sibling node is shown incident to its border, whereas the neighbourhoods of the component atoms of a sibling node are shown directly incident to the atoms. In subfigure (c) of figure 3.14, the Lambek edge whose target is $\mathcal{B}_3^-$ is a hyperedge whose source is the set $\{\mathcal{B}_7^+, \mathcal{B}_{5,8}^+\}$.

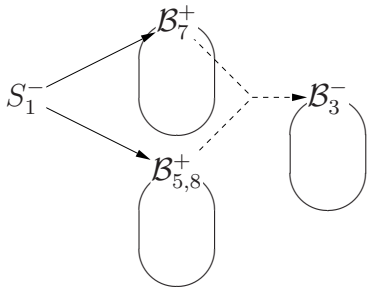
The two types of edges in an ATG have a kind of semantics analogous to the two

$$S_1/(N_2/N_3), S_4/(NP_5\backslash(NP_6/N_7))/NP_8, NP_9 \vdash S_{10}$$

(a) The sequent.



(b) A PTG for the sequent in 3.14a.



(c) The concise representative ATG for the PTG in 3.14b.

Figure 3.14: An example of a PTG and its concise representative ATG with a hyperedge.

types of edges in a PTG. Regular edges specify structure that has already been determined whereas Lambek edges specify paths that at some point must be completed. Lambek edges whose source is a set of positive sibling nodes require at least one path from a component atom of one of those sibling nodes to the target of the Lambek edge. Lambek edges whose target is a negative sibling node require a path to each of the component atoms of that sibling node. In other words, there is an existential quantifier on the source of Lambek edges in an ATG and a universal quantifier on the target of Lambek edges in an ATG. Lambek edges whose source is an atom have the same interpretation as Lambek edges in PTGs, namely that there must eventually be a path from that source. Similarly, Lambek edges whose target is an atom also have the same interpretation as Lambek edges in PTGs, namely that there must eventually be a path to that target.

Because of the relationship between sibling nodes and their component atoms, we will introduce a definition for a special kind of path in an ATG, in contrast to the standard definition of paths given in definition 3.1.5. The intuition is that we now want to be able to capture the relationship between sibling nodes and their component atoms within paths.

Definition 3.4.2. A *regular sibling path* P in an ATG $\hat{G}$ is a sequence of vertices $\langle v_1, \ldots, v_l \rangle$ such that $v_i \in \mathcal{V}(\hat{G})$ and for $1 \leq i < l-1$ either the edge $\langle v_i, v_{i+1} \rangle \in \mathcal{E}_{\mathcal{R}}(\hat{G})$ or v_i is a sibling node and v_{i+1} is a component atom of v_i. We will refer to the set of regular sibling paths of an ATG $\hat{G}$ as $\mathcal{P}_{\mathcal{RS}}(\hat{G})$.

As before, we will refer to v_1 as the *initial vertex* of P, v_l as the *final vertex* of P and the vertices $v_2, \ldots, v_{l-1}$ as the *intermediate vertices* of P. We will refer to the set consisting of the initial vertex and the final vertex as the *endpoints* of a path.

We will extend many of the usual graph-theoretic terms to regular sibling paths by prepending the term *regular sibling*. For example, we say that a graph is *regular sibling acyclic* if it has no regular sibling paths beginning and ending at the same vertex.

The next four subsections provide definitions for relating PTGs and ATGs and algorithms for manipulating ATGs in a variety of ways. Section 3.4.1 defines the relationship between PTGs and ATGs that will provide the foundation for the correctness of our algorithm. Section 3.4.2 gives algorithms for building ATGs for PTGs with large linkages out of ATGs for PTGs with smaller linkages. Section 3.4.3 defines conditions for ATGs that mirror the incremental integrity conditions for PTGs and an algorithm for determining whether an ATG satisfies those conditions. Finally, section 3.4.4 specifies an algorithm for removing vertices from an ATG that represents a PTG, while still maintaining that it represents that PTG, with the purpose of iteratively producing concise ATGs.

3.4.1 Representative ATGs

Our definition of ATGs gives us a data structure for storing information, but we are only interested in those ATGs that represent the structure of a PTG. In this section, we will introduce definitions that establish a correspondence between PTGs and ATGs. The edge-level correspondence will consist of a correspondence between regular paths in the PTG and regular sibling paths in the ATG and a correspondence between Lambek edges in the PTG and Lambek edges in the ATG. The graph-level correspondence holds if all of the regular sibling paths and Lambek edges in the ATG and all of the regular paths and Lambek edges in the PTG have correspondences.

Definition 3.4.3. Given an ATG $\hat{G}$ and a PTG G, an edge $\langle s, t\rangle \in \mathcal{E}_{\mathcal{L}}(\hat{G})$, where s is a set of positive vertices and t is a negative vertex, *represents* an edge $\langle s', t'\rangle \in \mathcal{E}_{\mathcal{L}}(G)$ iff the following hold:

1. If s is a singleton set consisting of a positive atom, then $s = \{s'\}$
2. If $s = \{s_1, \ldots, s_l\}$ is a set of positive sibling nodes, then there is a regular path in G from s' to the in-neighbour of s_i in the abstract proof frame for $1 \leq i \leq l$ and for each regular path from s to a positive vertex with regular out-degree 0 in G, there is a positive vertex p on that path such that $\mathcal{B}(p) \in s$
3. If t is a negative sibling node, then $t = \mathcal{B}(t')$
4. If t is a negative atom, then $t \neq t'$ and there is a regular path from t to t' in G

The intuition is that Lambek edges in ATGs represent Lambek edges in PTGs if they specify the vertices that have paths to and from the endpoints of the Lambek edge in the PTG.

In particular, the source of a Lambek edge in an ATG is either the source of the Lambek edge that it represents in the PTG or a set of positive sibling nodes that have

regular paths from the source of the Lambek edge that it represents in the PTG. Similarly, the target of a Lambek edge in an ATG is either the sibling node of the target of the Lambek edge that it represents in the PTG or a negative atom that has a regular path to the target of the Lambek edge that it represents in the PTG.

Definition 3.4.4. Given an ATG $\hat{G}$ and a PTG G, a regular sibling path $\langle v_1, \ldots, v_l \rangle \in \mathcal{P}_{\mathcal{RS}}(\hat{G})$ *represents* a regular path $\langle a_1, \ldots, a_m \rangle \in \mathcal{P}_{\mathcal{R}}(G)$ iff there is an injection ϕ from $(1, \ldots, l)$ to $(1, \ldots, m)$ such that the following hold for $1 \leq i \leq l$:

1. If v_i is an atom, then $v_i = a_{\phi(i)}$
2. If v_i is a sibling node, then $a_{\phi(i)} \in \mathcal{B}^{-1}(v_i)$
3. For $1 \leq j < i$, $\phi(j) < \phi(i)$

The intuition behind representative regular sibling paths in an ATG is similar to the intuition behind representative Lambek edges. In particular, a regular sibling path in an ATG represents a regular path in a PTG if each vertex in the regular sibling path has a corresponding vertex in the regular path, in order.

Then, based on these definitions, we can say what it means for an entire ATG to represent an entire PTG.

Definition 3.4.5. An ATG $\hat{G}$ *represents* a PTG G iff the following hold:

1. The atoms and sibling nodes that are peripheral to G are in $\mathcal{V}(\hat{G})$
2. For each edge in $\mathcal{E}_{\mathcal{R}}(\hat{G})$, there is a path in $\mathcal{P}_{\mathcal{R}}(G)$ that it represents
3. For each path $\langle v_1, \ldots, v_l \rangle \in \mathcal{P}_{\mathcal{R}}(G)$, if the following hold:
   (a) $v_1 \in \mathcal{V}(\hat{G})$
   (b) either $v_l \in \mathcal{V}(\hat{G})$ or v_l is positive and $\mathcal{B}(v_l) \in \mathcal{V}(\hat{G})$

   then there is a regular sibling path in $\mathcal{P}_{\mathcal{RS}}(\hat{G})$ that represents $\langle v_1, \ldots, v_l \rangle$

4. For each edge in $\mathcal{E}_{\mathcal{L}}(\hat{G})$, there is an edge in $\mathcal{E}_{\mathcal{L}}(G)$ that it represents

5. For each edge $\langle s, t\rangle \in \mathcal{E}_{\mathcal{L}}(G)$, such that there is no regular path from s to t in G, if the following hold:

   (a) either $s \in \mathcal{V}(\hat{G})$ or there is a positive sibling node $u \in \mathcal{V}(\hat{G})$ such that there is a regular path in G from s to the in-neighbour of u in the abstract proof frame

   (b) either $\mathcal{B}(t) \in \mathcal{V}(\hat{G})$ or there is a negative atom $a \in \mathcal{V}(\hat{G})$ such that there is a regular path in G from a to t

   then there is an edge in $\mathcal{E}_{\mathcal{L}}(\hat{G})$ that represents $\langle s, t\rangle$

Clause 1 of definition 3.4.5 specifies that the vertices at the periphery of a PTG must be in the ATG. As will be shown during the course of this chapter, vertices at the periphery of a partial term graph are the only vertices that are important for determining the integrity of a PTG. Our definition allows for additional vertices to appear in the vertex set of an ATG since we will need ATGs with some additional vertices for the various intermediate steps in our algorithm.

Clauses 2 and 4 specify that each regular edge in the ATG represents a regular path in the PTG, and that each Lambek edge in the ATG represents a Lambek edge in the PTG, respectively. Or, in other words, there are no edges in the ATG that do not represent something in the PTG.

Clauses 3 and 5 specify that if an edge or path in the PTG could be represented in the ATG, then it is. In general, if an edge or path cannot be represented, it is because the necessary endpoints for it to be represented are not present in the ATG. These clauses, together with the requirement that vertices that are peripheral to G must be in the vertex set of the ATG, combine to ensure that all of the paths and edges important for determining integrity are represented in a representative ATG. Note that in clause 2, the

universal quantifier is over edges and not paths. This simplifies our proofs and because paths are made up of edges, implies that all regular sibling paths in the ATG must also represent regular paths in the PTG.

Clause 5 contains the additional provision that if there is a regular path between the source and the target of a Lambek edge, then it is not required to be represented in a representative ATG. The intuition here is that such a Lambek edge has been satisfied according to the integrity requirements, so such an edge does not need to be represented to determine whether the PTG is integral. However, clause 4 allows such Lambek edges to be represented in the ATG anyway, because we will need such edges in the various intermediate steps in our algorithm.

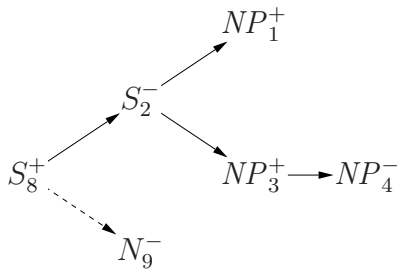
We should also note that an ATG for a term graph need not have any vertices at all, since there are no atom occurrences at the periphery of a non-partial term graph. In the chart parsing algorithm, we will need to look for empty ATGs in certain positions to determine whether any integral term graphs exist for a particular sequent.

Figures 3.15 and 3.16 depict PTGs and several representative ATGs for each. Subfigure (a) in each of those figures depicts the PTG and subfigures (b-e) depict representative ATGs for the PTG.

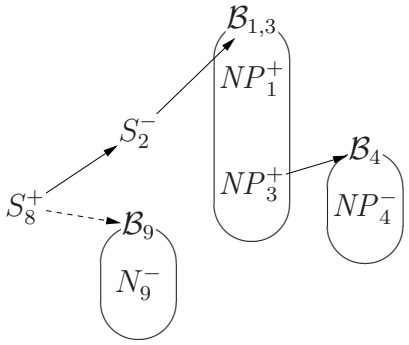
The preceding definition of representative ATGs gives us our correspondence between ATGs and PTGs, but the definition as given is more general than what will actually be inserted into the chart of our chart parsing algorithm. The purpose of our general definition above is to be able to use ATGs in intermediate forms, while we are processing ATGs in the chart. We will now define the ATGs which will actually be inserted into the chart:

Definition 3.4.6. A representative ATG $\hat{G}$ of a PTG G is *concise* iff $V(\hat{G})$ is exactly the atoms and sibling nodes that are peripheral to G.

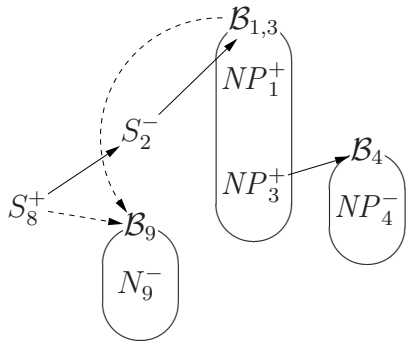
We will now provide an intuitive argument for the uniqueness of the concise representative ATG for a PTG. The key to the argument is that for each vertex in the ATG,



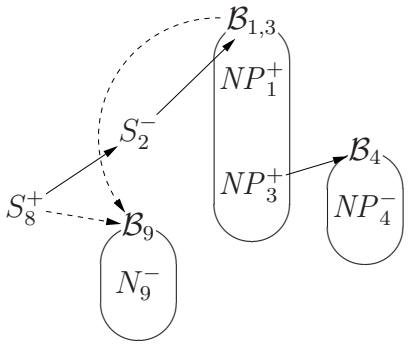
(a) The PTG in figure 3.12.



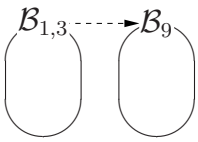
(b) After introduction of sibling nodes.



(c) Introduction of Lambek edges.



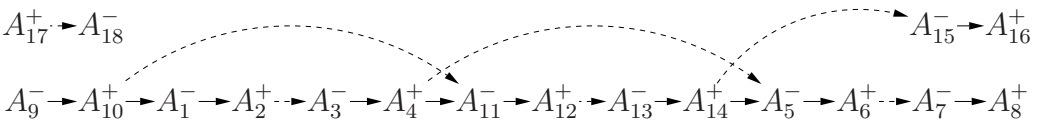
(d) Introduction of regular edges.



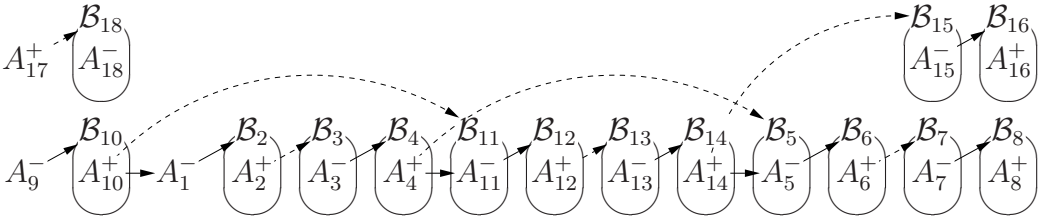
(e) The concise representative ATG.

Figure 3.15: Conversion of the PTG in figure 3.12 into its concise representative ATG.

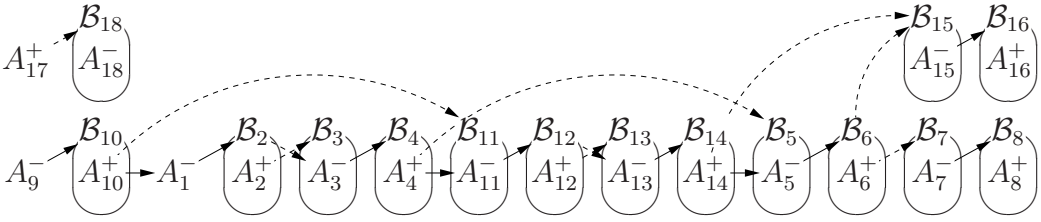
its neighbourhood is unique due to the representativeness requirements, and the fact that only peripheral atoms and sibling sets appear in the ATG. The representativeness requirements constrain the paths surrounding the vertices in the PTG. The requirement that exactly the peripheral atoms and sibling sets appear in the ATG constrains the lengths of paths in the ATG. We can then consider each condition in definition 3.4.5 and each type of vertex in an ATG to demonstrate the uniqueness of their neighbourhoods, which, in turn, establishes the uniqueness of the concise representative ATG.



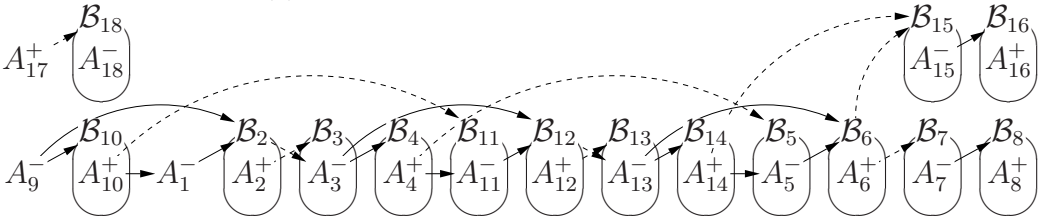
(a) The PTG in figure 3.13.



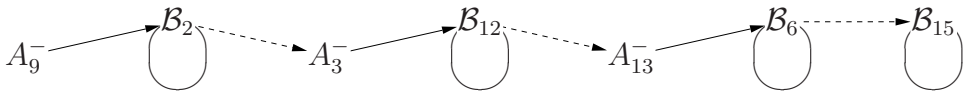
(b) After introduction of sibling nodes.



(c) After introduction of Lambek edges.



(d) After introduction of regular edges.



(e) Deletion of non-peripheral vertices to yield the concise representative ATG.

Figure 3.16: Conversion of the PTG in figure 3.13 into its concise representative ATG. Sibling nodes are identified by $\mathcal{B}$ and a subscript indicating their component atoms.

The existence of the concise representative ATG for a PTG is established either through algorithm 1 or through the vertex contraction algorithms in section 3.4.4.

Algorithm 1 converts an incrementally L^*-integral PTG to its concise representative

Algorithm 1 Convert PTG to concise representative ATG (G)

Copy G to $\hat{G}$
for each sibling node t in the abstract proof frame **do**
  Let s be the in-neighbour of t in the abstract proof frame
  Add t to $\hat{G}$
  Let $E = \mathcal{E}_{\mathcal{L}}(\hat{G})$ if t is positive and let $E = \mathcal{E}_{\mathcal{R}}(\hat{G})$ if t is negative
  Add $\langle s, t \rangle$ to E
  for Atom $a \in \mathcal{B}^{-1}(t)$ **do**
    Delete $\langle s, a \rangle$ from E
for each Lambek edge $\langle s, t \rangle \in G$ **do**
  if there is not a regular path from s to t in G **then**
    if s is open in G **then**
      Let s' be s
    else
      Let s' be the set of positive sibling nodes with regular sibling paths from s in $\hat{G}$
    if t is open in G **then**
      Let t' be $\mathcal{B}(t)$
    else
      Let t' be the unique negative atom that is open in G such that there is a regular path from t' to t in G
    Add $\langle s', t' \rangle$ to $\mathcal{E}_{\mathcal{L}}(\hat{G})$
for each regular sibling path in $\hat{G}$ from negative atom s to positive sibling node t **do**
  Add $\langle s, t \rangle$ to $\mathcal{E}_{\mathcal{R}}(\hat{G})$
Remove non-peripheral atoms and sibling nodes from the sources of Lambek edges
Delete all non-peripheral atoms and sibling nodes and their incident edges
return $\hat{G}$

ATG. It is not possible to convert an arbitrary PTG to a concise representative ATG, because arbitrary PTGs can contain regular cycles which cannot be represented in an concise representative ATG. This algorithm is only for expository purposes, and we will not use it in chart parsing because there are an exponential number of PTGs for an input and converting them would be too inefficient.

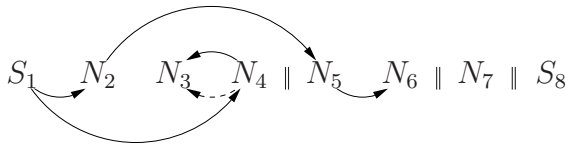
The intuition behind algorithm 1 is as follows. Line 1 copies the PTG to become the ATG $\hat{G}$. However, ATGs require sets of siblings to be represented as sibling nodes. Lines 2 through 8 introduce the sibling nodes and convert the relevant edges. Then, lines 9 through 21 introduce edges incident to the peripheral vertices that will represent the edges incident to non-peripheral vertices once the non-peripheral vertices are deleted in lines 22 and 23. Lines 9 through 19 do this for the Lambek edges and lines 20 and 21 do this for the regular edges. Line 10 determines whether the Lambek edge in G will be represented at all in $\hat{G}$. Lines 11 through 18 determine the sources and the target of the Lambek edge in $\hat{G}$. Then, lines 20 and 21 introduce regular edges into $\hat{G}$ for every path in G. This is done so that once all of the non-peripheral vertices are deleted, all of the structure represented by the edges incident to them is preserved. Finally, lines 22 and 23 delete the non-peripheral vertices and their incident edges.

Figures 3.15 and 3.16 show the application of the algorithm to two PTGs, yielding their respective concise representative ATGs.

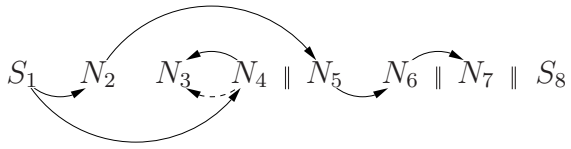
3.4.2 Expansion

The purpose of both ATGs and PTGs is to be able to incrementally determine the integrity of pieces of term graphs. Therefore, we will need to be able to build representative ATGs up incrementally from representative ATGs for PTGs with smaller linkages.

Because the axiomatic linkages for integral term graphs must be planar in the half plane, a PTG can be classified as one of two types: a bracketing PTG, or an adjoining PTG.



(a) A bracketing PTG.



(b) An adjoining PTG.

Figure 3.17: Partial term graphs for the sequent $(S/N)/(N/N), N/N, N \vdash S$.

Definition 3.4.7. A *bracketing* PTG is a PTG $G = \langle F, K \rangle$ with span $[a_l, a_r]$, where $\mathcal{K}_K(a_l) = a_r$. We say that a bracketing PTG G *extends* the PTG $\langle F, K' \rangle$, where K' is identical to K except that it does not contain the link between a_l and a_r. An *adjoining* PTG is a PTG $G = \langle F, K \rangle$ with span $[a_l, a_r]$, where $\mathcal{K}_K(a_l) \neq a_r$. We say that an adjoining PTG G *extends* a pair of PTGs $\langle \langle F, K' \rangle, \langle F, K'' \rangle \rangle$, where K' is a linkage whose leftmost vertex is a_l, K'' is a linkage whose rightmost vertex is a_r, both K' and K'' are sublinkages of K and the rightmost vertex of K' is adjacent to the leftmost vertex of K''.

Figure 3.17 depicts a bracketing and an adjoining PTG. The span for the bracketing PTG is $[N_2, N_5]$, and we can see that $\mathcal{K}(N_2) = N_5$. The PTG that this bracketing PTG extends is the one whose linkage consists solely of the link between N_3 and N_4. The span for the adjoining PTG in figure 3.17 is $[N_2, N_7]$. It extends the pair of PTGs consisting of the bracketing PTG shown in figure 3.17a and the PTG whose linkage consists of the single link between N_6 and N_7.

Having defined these two types of PTGs, we can now proceed with the methods for building representative ATGs for these PTGs out of the representative ATGs for the

PTGs that they extend.

Before we begin, we should discuss insertion into ATGs. Throughout the subsections on Bracketing and Adjoining, we will be inserting edges and vertices into various ATGs. Sometimes, edges and vertices that are already present in an ATG will be added to that ATG. In these cases, the edges and vertices within a graph are not duplicated and the insertions will do nothing.

Bracketing

A bracketing PTG is built from the PTG that it extends and the link between the leftmost and rightmost atoms in its span. Since we will be working with ATGs that represent those PTGs, we will need to specify the conditions on the ATG that determine whether extending its PTG yields an L^*-integral PTG, and we will need to consider how to construct an ATG that represents the bracketing PTG.

Let G be a bracketing PTG over span $[a_l, a_r]$ and let H be the PTG that G extends. We begin with an algorithm that determines whether bracketing can occur at all. According to the definition of term graphs in section 3.2, each link must match atom occurrences of opposite polarity that have the same atom.

Algorithm 2 LinkEligibility(a_l, a_r)

```
1: if the polarities of a_l and a_r are the same then
2:     return False
3: if the atoms of a_l and a_r are not the same then
4:     return False
5: return True
```

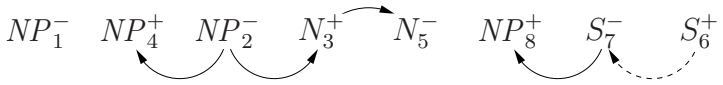
If algorithm 2 returns True on the two atom occurrences a_l and a_r, then we can proceed to build a representative ATG for G from the concise representative ATG for H. Let $\hat{G}$ be an ATG for G and let $\hat{H}$ be the concise representative ATG for H. Then, algorithm 3 specifies how to build $\hat{G}$ from $\hat{H}$.

Algorithm 3 Bracketing($\hat{H}$, a_l, a_r)

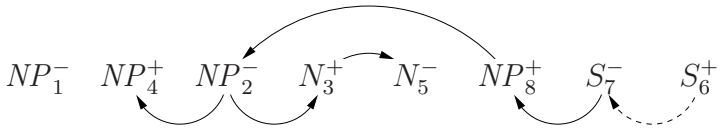
Let $\hat{G}$ be a copy of $\hat{H}$
if $\hat{G}$ is empty **then**
    Let Λ be the set consisting of the left split of a_l, the right split of a_l, the left split of a_r and the right split of a_r
    Let $\Xi = \emptyset$
else
    Let Λ be the set consisting of the left split of a_l and the right split of a_r
    Let Ξ be the set consisting of the right split of a_l and the left split of a_r
for $\lambda \in \Lambda$ **do**
    Let S be the set of edges in $CEA_F(\lambda) - \cup_{\xi \in \Xi} CEA_F(\xi)$
    for each $\langle s, t \rangle \in S$ **do**
        Insert s and t into $V(\hat{G})$
        if s is negative **then**
            Insert the edge $\langle s, t \rangle$ into $E_R(\hat{G})$
        else
            Insert the edge $\langle \{s\}, t \rangle$ into $E_L(\hat{G})$
Insert a_l and a_r into $V(\hat{G})$
Let a_p be the positive vertex of a_l and a_r and let a_n be the negative
Insert the edge $\langle a_p, a_n \rangle$ into $E_R(\hat{G})$
return $\hat{G}$

$$NP_1, NP_2/N_3\backslash NP_4, N_5 \vdash S_6/(S_7\backslash NP_8)$$

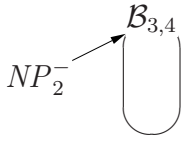
(a) The sequent.



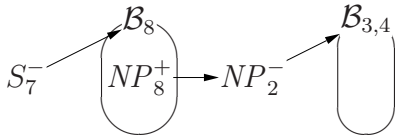
(b) The PTG prior to extension.



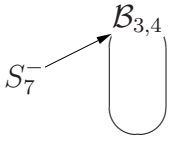
(c) The bracketing PTG.



(d) The ATG for the PTG in (b).



(e) The ATG after Bracketing.



(f) The ATG after vertex contraction.

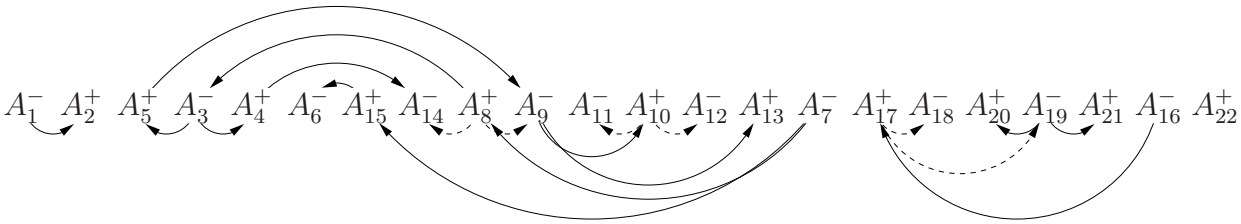
Figure 3.18: A linguistically motivated example of the process for bracketing an ATG.

Correctness In this section we will provide an intuitive argument for the correctness of algorithm 3, which means arguing that it returns a representative ATG for G.

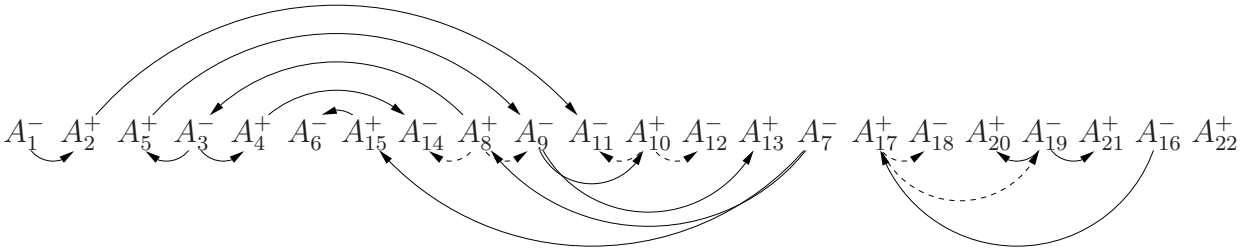
First, algorithm 3 determines all of the vertices that must be copied into $\hat{G}$ from the abstract proof frame to satisfy the first clause of definition 3.4.5, and then we also copy all of the edges incident to those vertices. Then, we add an edge from the positive atom occurrence to the negative atom occurrence because this edge is introduced in the PTG via the new linkage edge. Λ represents the set of splits whose crossing edges must be introduced to the ATG. Ξ represents the set of splits that have already been introduced, so their crossing edges should be explicitly excluded from introduction.

$$A_1/A_2, A_3/A_4\backslash A_5, A_6, A_7\backslash(A_8\backslash(A_9/(A_{10}/A_{11}\backslash A_{12})/A_{13})/A_{14})\backslash A_{15}, A_{16}\backslash(A_{17}\backslash A_{18}\backslash(A_{19}\backslash A_{20}/A_{21})) \vdash A_{22}$$

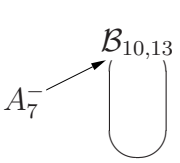
(a) The sequent.



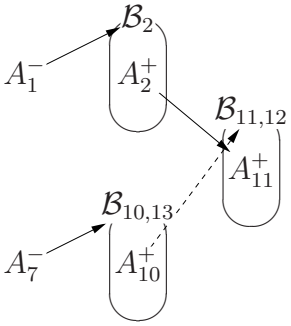
(b) The extended PTG.



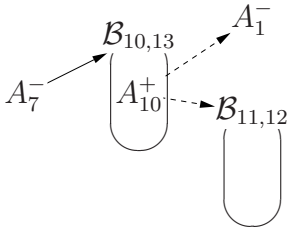
(c) The bracketing PTG.



(d) The ATG for the PTG in (b).



(e) The ATG after Bracketing.



(f) The ATG after vertex contraction.

Figure 3.19: An example of the process for bracketing an ATG.

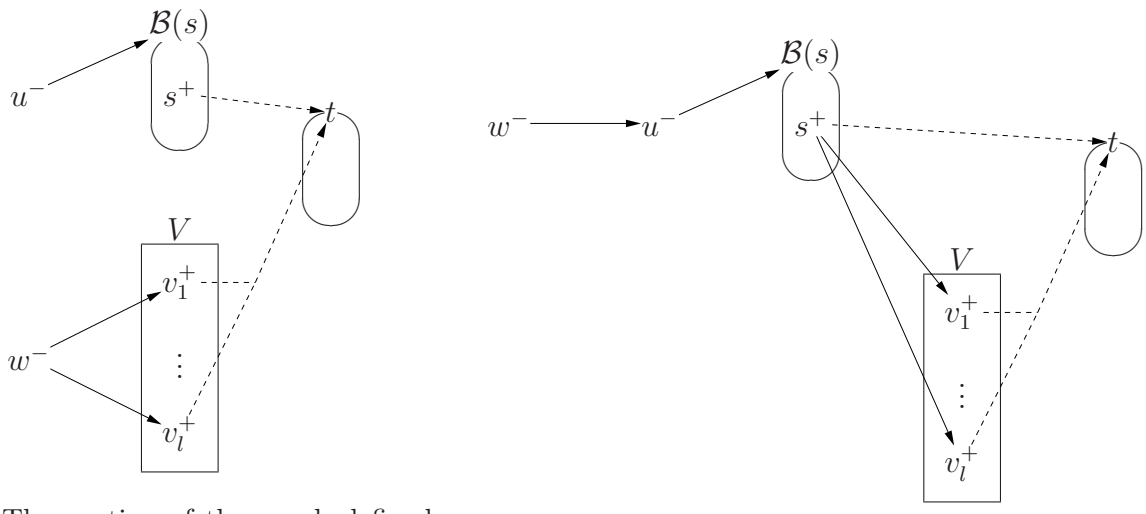
Figures 3.18 and 3.19 show the application of algorithm 3 and the subsequent vertex contraction.

Computational Complexity In the previous section, we argued that algorithm 3 correctly produces the ATG for a bracketing PTG, where the input was the ATG for the PTG that it extends and did not include the PTG itself. In this section, we will analyze its running time. Throughout these proofs, we let n be the number of atoms in the input and k be the maximum order of categories in the input. Then, by proposition 3.1.6, which proves that the number of vertices in the abstract proof frame crossing a split is bounded by the order of the category, and the conciseness of $\hat{H}$, we know that the number of vertices in $\hat{H}$ is bounded by $2k$, or twice the maximal order of the categories in the lexicon.

Line 1 requires a complete copy of $\hat{H}$ which has $2k$ vertices. Since $\hat{H}$ is an ATG, its regular subgraph is a simple graph and its Lambek subgraph is a simple hypergraph[9]. This means that there are $O(k^2)$ regular edges and $O(k2^{2k})$ Lambek edges. Therefore, copying $\hat{H}$ into $\hat{G}$ takes $O(k2^{2k})$ time. Then, lines 2 through 7 can be done in constant time, since calculating splits for atoms is a simple lookup. Line 10 through 15 iterate over the edges in the abstract proof frame crossing each split which we know is bounded by k. Each iteration of the loop is constant time since we are only inserting vertices and edges into a graph. Line 8 iterates over the number of splits in Λ, which can be as many as 4 which means that the loop at line 8 can be performed in $O(k)$ time. Finally, lines 16 through 19 are all constant time operations.

Then, based on our assumptions, the bracketing algorithm given in algorithm 3 can be performed in $O(k2^{2k})$ time.

[9] A *simple hypergraph* is a hypergraph with no loops and whose edges are a set rather than a multiset.



(a) The portion of the graph defined on lines 15 through 18.

(b) The portion of the graph built on lines 19 through 25.

Figure 3.20: The graphs built on lines 15 through 25 in algorithm 4.

Adjoining

An adjoining PTG is built out of two smaller PTGs by combining their linkages. As in the case of bracketing, we will be working with representative ATGs of these PTGs. As such, we will need to specify an algorithm for building a representative ATG for an adjoining PTG out of the concise representative ATGs for the two PTGs that it extends.

Let F be a proof frame and let G be an adjoining PTG over F with span $[a_l, b_r]$, let $\langle G_1, G_2 \rangle$ be a pair of PTGs that G extends, let $[a_l, a_r]$ be $\mathcal{N}(G_1)$ and let $[b_l, b_r]$ be $\mathcal{N}(G_2)$. Let $\hat{G}_i$ be a concise representative ATG for G_i for $1 \leq i \leq 2$. Finally, let T be the right split of G_1 (which is the same as the left split of G_2). Then, algorithm 4 specifies how to build an adjoining ATG.

Correctness In this section we will provide an intuitive argument for the correctness of algorithm 4, which means arguing that it returns a representative ATG for G.

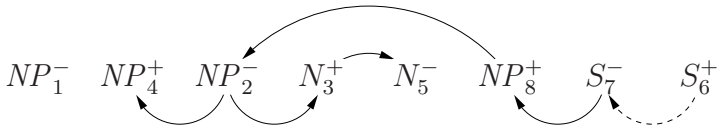
The intuition behind algorithm 4 is that we are combining the edges in ATGs $\hat{G}_1$ and

Algorithm 4 Adjoining(F, $\hat{G}_1$, $\hat{G}_2$, T)

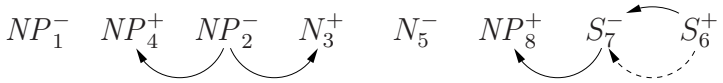
Let $\hat{G} = \langle \mathcal{V}(\hat{G}_1) \cup \mathcal{V}(\hat{G}_2), \mathcal{E}_{\mathcal{R}}(\hat{G}_1) \cup \mathcal{E}_{\mathcal{R}}(\hat{G}_2), \mathcal{E}_{\mathcal{L}}(\hat{G}_1) \cup \mathcal{E}_{\mathcal{L}}(\hat{G}_2), \mathcal{N}(\hat{G}_1) + \mathcal{N}(\hat{G}_1) \rangle$
for $\langle s, t \rangle \in CE_F(T)$ ordered topologically **do**
  if $\langle s, t \rangle$ is regular **then**
    if t has an in-neighbour in $\hat{G}$ that is not s **then**
      Let u be the regular in-neighbour of t in $\hat{G}$ that is not s
      Delete $\langle u, t \rangle$ from $\mathcal{E}_{\mathcal{R}}(\hat{G})$
      Insert $\langle u, s \rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$
    Insert $\langle s, t \rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$
  else
    if s is not open in G **then**
      if s has no in-neighbour in F **then**
        Let U be the set of positive sibling nodes with no regular in-neighbour in $\hat{G}$
        Insert $\langle s, u \rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$ for each $u \in U$
      else
        Let u be the in-neighbour of $\mathcal{B}(s)$ in $APF(F)$
        Let V be the Lambek in-neighbour in $\hat{G}$ of t that is not s
        Let w be the regular in-neighbour in $\hat{G}$ of the vertices in V
        Let $i = 1$ if s is not open in G_1 and let $i = 2$ if s is not open in G_2
        if u is not open in G_i **then**
          Insert u and $\mathcal{B}(s)$ into $\mathcal{V}(\hat{G})$
          Insert $\langle u, \mathcal{B}(s) \rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$
          Insert $\langle w, u \rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$
        for $v \in V$ **do**
          Delete $\langle w, v \rangle$ from $\mathcal{E}_{\mathcal{R}}(\hat{G})$
          Insert $\langle s, v \rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$
      Insert $\langle s, t \rangle$ into $\mathcal{E}_{\mathcal{L}}(\hat{G})$
return $\hat{G}$

$$NP_1, NP_2/N_3\backslash NP_4, N_5 \vdash S_6/(S_7\backslash NP_8)$$

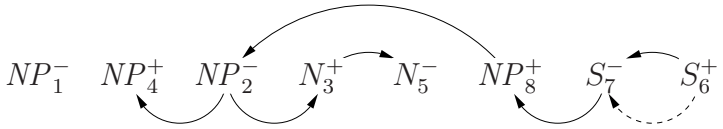
(a) The sequent.



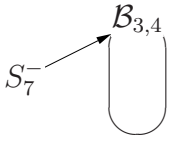
(b) The right-extended PTG.



(c) The left-extended PTG.

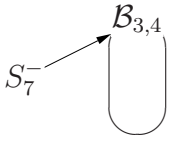


(d) The adjoining PTG.

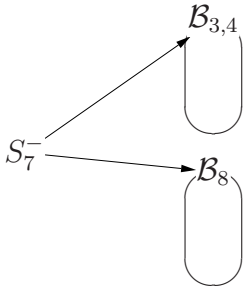


(e) The concise representative ATG for the PTG in 3.21b.

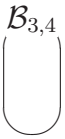
(f) The concise representative ATG for the PTG in 3.21c.



(g) The ATG before any Adjoining iterations.



(h) The ATG after Adjoining.

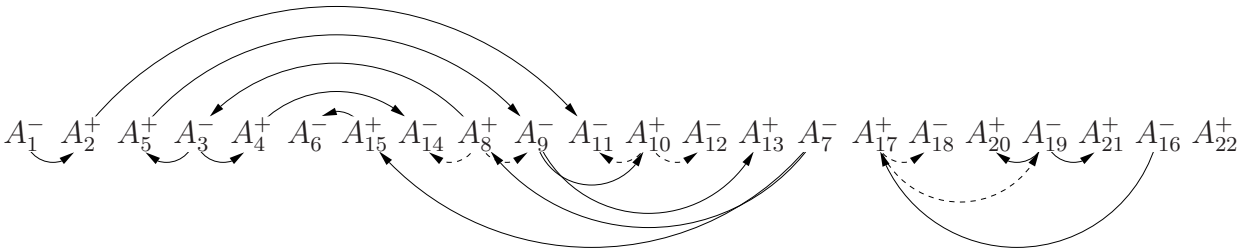


(i) The ATG after vertex contraction.

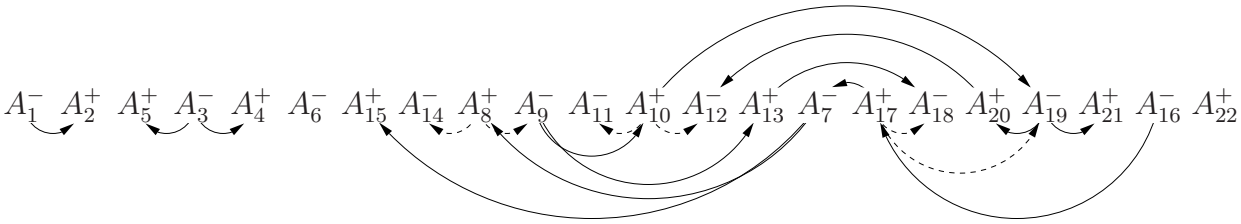
Figure 3.21: A linguistically motivated example of the process for adjoining two ATGs.

$$A_1/A_2, A_3/A_4\backslash A_5, A_6, A_7\backslash(A_8\backslash(A_9/(A_{10}/A_{11}\backslash A_{12})/A_{13})/A_{14})\backslash A_{15}, A_{16}\backslash(A_{17}\backslash A_{18}\backslash(A_{19}\backslash A_{20}/A_{21}))\vdash A_{22}$$

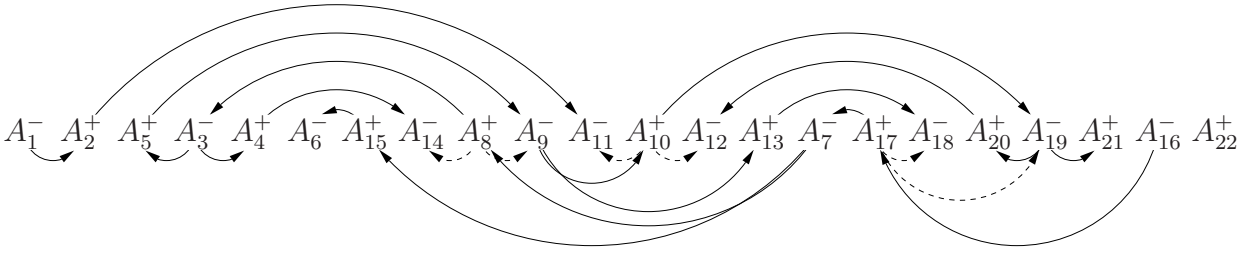
(a) The sequent.



(b) The right-extended PTG.

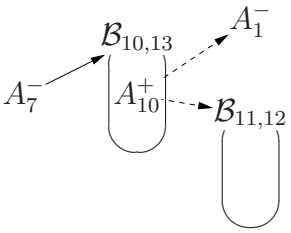


(c) The left-extended PTG.

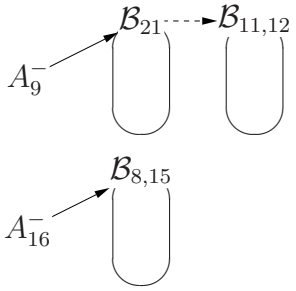


(d) The adjoining PTG.

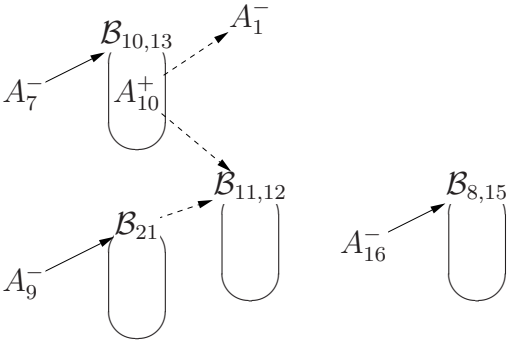
Figure 3.22: An example sequent and PTGs for adjoining.



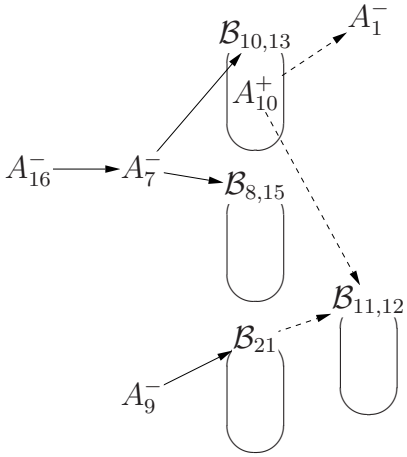
(a) The concise representative ATG for the PTG in 3.22b.



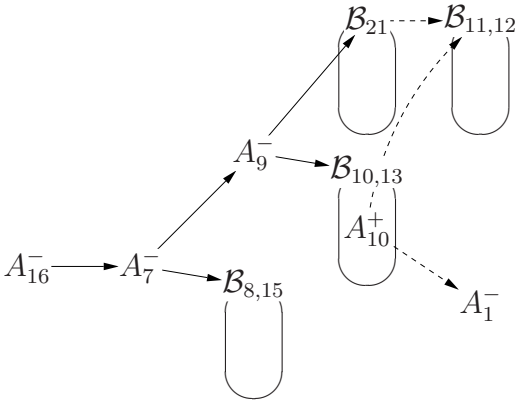
(b) The concise representative ATG for the PTG in 3.22c.



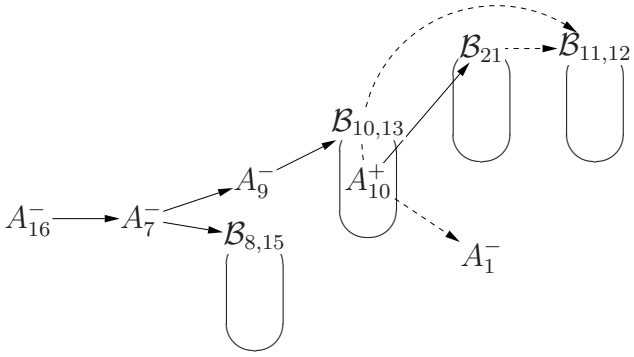
(c) The ATG before any Adjoining iterations.



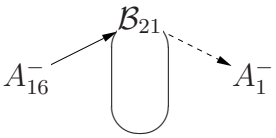
(d) The ATG after 1 Adjoining iterations.



(e) The ATG after 2 Adjoining iterations.



(f) The ATG after Adjoining.



(g) The ATG after vertex contraction.

Figure 3.23: An example of the process for adjoining two ATGs.

$\hat{G}_2$ and then including those edges that cross the split between those two. The regular edges crossing that split indicate structure that is already present in both $\hat{G}_1$ and $\hat{G}_2$, so the edges must be reoriented in lines 6 through 8. The Lambek edges crossing that split specify paths that must eventually exist, and so are simply inserted into $\hat{G}$ as is. In addition, the Lambek edges crossing T give us information about the location of their sources in $\hat{G}$, which indicates the structure of some of the out-neighbourhoods of negative atoms in $\hat{G}$.

The edges crossing the split T are considered according to whether they are regular edges or Lambek edges. The regular edges are considered in lines 3 through 8. Such regular edges necessarily appear in the PTG G, and using this information, along with where their endpoints are in $\hat{G}_1$ and $\hat{G}_2$, we can reconstruct the required structure in $\hat{G}$. The Lambek edges are considered in lines 10 through 26. These edges indicate the location of their endpoints in $\hat{G}_1$ and $\hat{G}_2$ in a way that is analogous to the regular edge case. Lines 12 and 13 consider the case where the Lambek edge has the unique positive atom with in-degree 0 in the proof frame. Then, lines 15 through 25 consider the general case. The structure of $\hat{G}$ near the Lambek edge is determined from $\hat{G}_1$ and $\hat{G}_2$ and edges are inserted and deleted as necessary.

We specify the order of iteration over the crossing edges on line 2 because this reduces the number of cases that need to be considered since the descendants of the vertices being processed are fixed. Also, keep in mind that the PTGs that we are considering all have linkages that are planar in the half plane and that are contiguous.

Figure 3.20 show the graphs constructed in the latter half of algorithm 4 on ATGs in general. Figures 3.21, 3.22 and 3.23 show the application of algorithm 4 and the subsequent vertex contraction.

Computational Complexity We have argued for the correctness of algorithm 4, and in this section we will prove that although algorithm 4 is exponential in the worst case

in general, it is constant time if the order of categories is bounded by a constant.

Line 1 of algorithm 4 requires the copying of each of the component sets of two ATGs. Since those ATGs are concise, the number of vertices is at most $2k$. Since the regular edge set is a set of distinct ordered pairs of vertices, there are at most $O(k^2)$ regular edges. Also, since the Lambek edge set is a set of pairs where the first element in the pair is a set of vertices and the second element is a vertex, there are $O(k2^{2k})$ Lambek edges. Therefore, line 1 takes time $O(k2^{2k})$ time.

Line 2 iterates over the edges crossing T, of which there are at most k, by proposition 3.1.6. Then, everything in lines 3 through 26 is constant time, except for the loop on lines 23 through 25. But, that loop is an iteration over vertices, so is bounded by $2k$.

Therefore, the entire algorithm runs in $O(k^2 2^{2k})$.

3.4.3 Incremental Integrity of Abstract Term Graphs

In section 3.3.1, we defined incremental integrity for PTGs and since ATGs are an abstraction over PTGs, we will need to extend the notion of incremental integrity to ATGs and provide an algorithm for checking the incremental integrity of ATGs.

Definition 3.4.8. Given an ATG $\hat{G}$, we define the following conditions on it:

$\mathbf{IT_A}(1)$: $\hat{G}$ is regular sibling acyclic

$\mathbf{IT_A}(2)$: For each $\langle S, t\rangle \in \mathcal{E}_{\mathcal{L}}(\hat{G})$, the following hold:

1. if t is a sibling node and t has at least one component atom not in $\mathcal{N}(\hat{G})$, then there exists a peripheral positive vertex $p \in \mathcal{V}(\hat{G})$ such that there is a regular sibling path from some $v \in S$ to p
2. if t is an atom, let T be the singleton set consisting of t and if t is a sibling node, let T be the set of component atoms of t in $\mathcal{V}(\hat{G})$, and then for $t' \in T$, one of the following hold:

(a) There is a regular sibling path whose initial vertex is some $v \in S$ and whose final vertex is t'

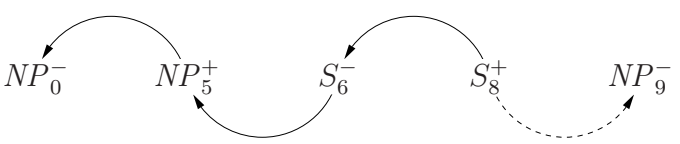
(b) There exists a peripheral positive vertex $p \in \mathcal{V}(\hat{G})$ such that there is a regular sibling path from some $v \in S$ to p and, there exists a peripheral negative atom $a \in \mathcal{V}(\hat{G})$ such that there is a regular sibling path from a to t'

We say that an ATG that satisfies $\mathbf{IT_A}(1)$ and $\mathbf{IT_A}(2)$ is *incrementally L^*-integral*. Figure 3.24 depicts a PTG that is not incrementally L^*-integral and a representative ATG for it that is not incrementally L^*-integral. The ATG in that figure is the result of adjoining the concise representative ATGs for the two bracketing PTGs. The ATG in figure 3.24 is not incrementally L^*-integral due to violating clause 2b of $\mathbf{IT_A}(2)$ because there is no regular sibling path from $\mathcal{B}_5$ to any peripheral positive vertex and NP_9 is a component atom of $\mathcal{B}_9$ and not in the span of the ATG.

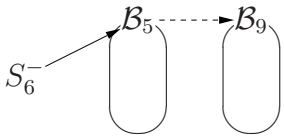
The incremental integrity conditions for PTGs and ATGs are quite similar. $\mathbf{IT_A}(1)$ is identical to $\mathbf{IT_P}(1)$, except that the prohibition of cycles has been extended to regular sibling paths to accommodate sibling nodes in ATGs. $\mathbf{IT_A}(2)$ and $\mathbf{IT_P}(2)$ are also quite similar, with the difference being in the treatment of sibling nodes in ATGs. Lambek edges where the target is an atom have identical requirements in $\mathbf{IT_A}(2)$, relative to $\mathbf{IT_P}(2)$, except that regular sibling paths have replaced regular paths. Sibling nodes that are the targets of Lambek edges are treated specially in two ways. First, the component atoms of sibling nodes appearing in $\hat{G}$ each behave as though they are the targets of the Lambek edges themselves. Then, there is an additional clause in $\mathbf{IT_A}(2)$ for Lambek edges where the target is a sibling node. This additional clause requires that sibling nodes with at least one component atom not in the span of $\hat{G}$ have paths from the source to a peripheral positive atom. The intuition here is that Lambek edges in ATGs with targets that are sibling nodes correspond to sets of Lambek edges in PTGs. Such Lambek edges in ATGs whose component atoms are all in the span are represented by other Lambek

$$NP_0, S_6\backslash NP_5 \vdash S_8\backslash NP_9$$

(a) The sequent.



(b) The PTG.



(c) The ATG after adjoining.

Figure 3.24: An ATG that is not incrementally L^*-integral.

edges with atoms as targets. Such Lambek edges in ATGs whose component atoms are not all in the span must still satisfy the requirement that there is a regular sibling path from the source to a peripheral positive vertex.

Despite defining integrity conditions for both term graphs and incremental integrity conditions for PTGs, we will not actually be checking them algorithmically. Instead, we will be checking these conditions by checking whether ATGs are incrementally integral which is simply a matter of iterating over the right edges and vertices.

Correctness

In this section, we will prove that their is a kind of equivalence between the incremental integrity conditions on PTGs and the incremental conditions on ATGs. We begin with the following proposition, proving one direction of the equivalence between the incrementally integrity conditions of PTGs and ATGs.

Proposition 3.4.9. *Given an incrementally L^*-integral PTG G, and a representative ATG $\hat{G}$ for G, $\hat{G}$ is incrementally L^*-integral.*

Proof. The key insight to this proof is that the incremental integrity conditions for PTGs and the incremental integrity conditions for ATGs are very similar and ATGs represent

a subset of the structure found in PTGs, so the ATG conditions are implied by the PTG conditions.

We will prove the lemma considering each of the incremental integrity conditions separately.

$\mathbf{IT_A}(1)$: Suppose on the contrary that $\hat{G}$ has a regular sibling cycle C. Then, because $\hat{G}$ represents G, there must be a regular cycle that C represents, which violates $\mathbf{IT_P}(1)$.

$\mathbf{IT_A}(2)$: Suppose on the contrary that $\hat{G}$ violates $\mathbf{IT_A}(2)$ and let $\langle S, t\rangle$ be a violating edge.

If t is an atom, let $\langle s, t\rangle$ be its Lambek in-edge in G. Since there is no regular sibling edge with initial vertex $v \in S$ and final vertex t, there cannot be a regular path in G from s to t. And, since there is not both a peripheral positive vertex with a regular sibling path in $\hat{G}$ from some $v \in S$ and a peripheral negative atom with a regular sibling path in $\hat{G}$ to t, there cannot be both a regular path in G from s to an open positive vertex and a regular path in G from an open negative vertex to t. So, G cannot satisfy $\mathbf{IT_P}(2)$, which is a contradiction.

If t is a sibling node with at least one component atom not in $\hat{G}$, the violation is due to the first clause. Let a be that component atom and let $\langle s, a\rangle \in \mathcal{E}_{\mathcal{L}}(G)$ be its Lambek in-edge in G. Then, since there is no regular sibling path in $\hat{G}$ from $v \in S$ to a peripheral positive vertex, there cannot be a regular path from s to an open positive vertex in G, which means that the second clause of $\mathbf{IT_P}(2)$ does not hold, and since t is not in the span of G, the first clause of $\mathbf{IT_P}(2)$ does not hold. Then, G does not satisfy $\mathbf{IT_P}(2)$, which is a contradiction.

If t is a sibling node, the violation is due to the second clause. Let t' be the violating component atom. Then, by reasoning identical to the case where t is an atom, G cannot satisfy $\mathbf{IT_P}(2)$, which is a contradiction.

□

We cannot simply prove the inverse of proposition 3.4.9, because ATGs are an abstraction over PTGs that omit various kinds of information. Instead, we must prove the inverse of proposition 3.4.9 on only the ATGs that are built by the Bracketing and Adjoining algorithms.

Proposition 3.4.10. *Let G be a bracketing PTG with span $\mathcal{N}(G) = [a_l, a_r]$ that extends an incrementally L^*-integral PTG H and let $\hat{H}$ be the concise representative ATG of H. Then, let $\hat{G}$ be the output of Bracketing($\hat{H}$, a_l, a_r) and let $\hat{G}$ be incrementally L^*-integral. Then G is incrementally L^*-integral.*

Proof. The key insight to this proof is that the ATGs produced by the bracketing algorithm preserve enough structure to ensure that checking the ATG incremental integrity conditions on them guarantees that the PTG incremental integrity conditions on G are satisfied.

We will prove this considering each of the incremental integrity conditions independently.

> $\mathbf{IT_P}(1)$: Suppose on the contrary that G contains a regular cycle C. Since H is incrementally L^*-integral, we know that it contains no regular cycle. Then, C must contain the regular edge between a_l and a_r, since G and H are identical except for this edge. Then, the edge inserted into $\hat{G}$ at line 18 of the Bracketing algorithm represents that edge. However, the regular path consisting of the other edges in C must have a representative regular sibling path P in $\hat{H}$, since its endpoints are peripheral to H. Then, since $\hat{H}$ represents H and since edges are not deleted in the Bracketing algorithm, P must appear in $\hat{G}$. But then a regular sibling cycle consisting of P and the representative in $\hat{G}$ of the regular edge between a_l and a_r appears in $\hat{G}$ and since $\hat{G}$ is regular sibling acyclic, we have a contradiction. Therefore, G is regular acyclic.

$\mathbf{IT_P}(2)$: Suppose on the contrary that G does not satisfy $\mathbf{IT_P}(2)$ and let $\langle s, t\rangle \in \mathcal{E}_{\mathcal{L}}(G)$ be a non-satisfying Lambek edge. Since H satisfies $\mathbf{IT_P}(2)$, and the only difference between G and H is the edge between a_l and a_r, it must be the case that there is no regular path from s to t in either of G or H. So, $\langle s, t\rangle$ violates either the first clause or clause 2 of $\mathbf{IT_P}(2)$. Then, since H is incrementally L^*-integral and there is no regular path in H from s to t, there must be an open positive vertex s' in H such that there is a regular path in H from s to s' and an open negative vertex t' in H such that there is a regular path in H from t' to t. These regular paths in H must also appear in G, since the only difference between G and H is the addition of the edge between a_l and a_r. Then, since G does not satisfy $\mathbf{IT_P}(2)$, either s' or t' must not be open in G, which means that one of them is one of a_l or a_r.

If s' is one of a_l or a_r, then, since $\hat{G}$ is incrementally L^*-integral, there must be a peripheral positive sibling set b for which there is a regular sibling path in $\hat{G}$ from s'. But, since $\hat{G}$ represents G, there must be a regular path in G from s to an open component atom of b, which is a contradiction. The proof for the case where t' is one of a_l or a_r is similar.

□

Proposition 3.4.11. *Let F be a proof frame. Then, let G be an adjoining PTG with proof frame F that extends a pair of PTGs $\langle G_1, G_2\rangle$ that are incrementally L^*-integral. Then, let $\hat{G}_1$ and $\hat{G}_2$ be the concise representative ATGs for G_1 and G_2, respectively. Let $\hat{G}$ be the output of Adjoining(F, $\hat{G}_1$, $\hat{G}_2$, T) where T is the split between G_1 and G_2 and let $\hat{G}$ be incrementally L^*-integral. Then, G is incrementally L^*-integral.*

Proof. The key insight to this proof is essentially the same as that for the bracketing algorithm. Again, we will prove the result for each incremental integrity condition independently.

$\mathbf{IT_P}(1)$: Assume on the contrary that G contains a regular cycle C. Since G_1 and G_2 are incrementally L^*-integral, they cannot contain any regular cycles, so C must contain edges in both G_1 and G_2. We will consider the edges in C according to whether they are an edge crossing T or not. Let P be a maximal subpath of C not containing any edges crossing T. Then, P must exist in one of G_1 or G_2. Furthermore, its endpoints are peripheral to at least one of G_1 or G_2, and since $\hat{G}$ represents G, there must be a regular sibling path in $\hat{G}$ that represents P. In addition, the edges in C that cross T have a source and target that are also peripheral to at least one of G_1 or G_2, so there must be a regular sibling edge in $\hat{G}$ that represents them. Then, these regular sibling paths form a regular sibling cycle in $\hat{G}$, which contradicts our assumption.

$\mathbf{IT_P}(2)$: Assume on the contrary that G does not satisfy $\mathbf{IT_P}(2)$ and let $\langle s, t\rangle \in \mathcal{E}_{\mathcal{L}}(G)$ be a non-satisfying Lambek edge. Consider the edge $\langle S', t'\rangle \in \mathcal{E}_{\mathcal{L}}(\hat{G})$ that represents $\langle s, t\rangle$. If there is a regular sibling path in $\hat{G}$ from a vertex in S' to a positive peripheral vertex and there is a regular sibling path in $\hat{G}$ from a peripheral negative atom to t', then since $\hat{G}$ represents G, there must be a regular path in G from s to an open positive vertex and a regular path in G from an open negative vertex to t. Therefore, $\hat{G}$ cannot satisfy $\mathbf{IT_A}(2)$, which is a contradiction.

□

Computational Complexity

Definition 3.4.8 introduced the incremental integrity conditions for ATGs and we will be repeatedly checking these to determine whether ATGs are incrementally integral. In this section, we will prove a time bound on checking these conditions.

We do not need to check the incremental integrity of ATGs on arbitrary ATGs, but rather only on ATGs that are the result of either the Bracketing or Adjoining algorithms.

Such ATGs have a maximum of $4k$ vertices since they are built from a maximum of two concise ATGs, in the case of the Adjoining algorithm. Then, since the regular edge set is a set of ordered pairs of vertices, there are at most $(4k)^2$ regular edges. Then, since the Lambek edge set is a set of hyperedges there are at most $2^{4k}4k$ Lambek edges.

Given these bounds, we know that we can check $\mathbf{IT_A}(1)$ in time $O(2^{4k}k)$ by a simple search. Checking $\mathbf{IT_A}(2)$ is somewhat more involved. There are $O(2^{4k}k)$ Lambek edges, and for each Lambek edge, we must determine the sets of vertices which have regular sibling paths from vertices in its source set, and the sets of vertices which have regular sibling paths to its target. Both of these sets can be determined by a simple graph search, each of which could take time $O(2^{4k}k)$ resulting in a total time of $O(2^{8k}k^2)$ for checking $\mathbf{IT_A}(2)$. In addition, for the Lambek edges whose target is a sibling node, we must determine whether there are component atoms of the target that are not in $\mathcal{N}(\hat{G})$ and which of the atoms in $V(\hat{G})$ are component atoms of the target. The former can be done in constant time by comparing the boundaries of the sibling node with $\mathcal{N}(\hat{G})$ and the latter can be done in time $O(k)$ by checking each atom in $V(\hat{G})$.

Combining these two times yields a total time of $O(2^{8k}k^2)$ for checking the incremental integrity of an ATG, which is constant, if k (the maximum order of the categories) is constant.

3.4.4 Vertex Contraction

In this section, we will introduce the algorithms for deleting non-peripheral vertices from an ATG. The contraction of vertices in an ATG is necessary for obtaining a *concise* representative ATG from a representative ATG. The general idea will be to contract the paths surrounding a vertex, so that any paths going through that vertex are maintained after that vertex has been deleted from the ATG.

Vertices are contracted after an ATG has been determined to be incrementally integral. Therefore, we can assume that the input ATGs are incrementally integral. Four

types of vertices exist in ATGs and we will need methods for contracting those vertices: one for positive atoms, one for negative atoms and one for both positive sibling nodes and negative sibling nodes. We will contract atoms before sibling nodes, so when a sibling node is deleted, none of the atoms belonging to it will be in the vertex set of the ATG.

The algorithms for contracting these four types of vertices are shown as algorithms 5, 6 and 7.

Correctness

In this section, we will prove that the vertex contraction algorithms shown in algorithms 5, 6 and 7 preserve the representativeness of the ATGs that they are manipulating. The idea is that we will begin with an ATG $\hat{G}$ that represents some PTG G and each application of a vertex contraction algorithm will delete a vertex from $\hat{G}$ to produce a new ATG $\hat{G}'$ that also represents G.

Algorithm 5 ContractPositiveAtom($\hat{G}$, a)

1: **if** $\mathcal{B}(a) \in \mathcal{V}(\hat{G})$ and there is an edge $\langle b, \mathcal{B}(a)\rangle \in \mathcal{E}_{\mathcal{R}}(\hat{G})$ **then**
2: **for** each $\langle a, c\rangle \in \mathcal{E}_{\mathcal{R}}(\hat{G})$ **do**
3: Insert $\langle b, c\rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$
4: **for** each edge $\langle \{a\}, b\rangle \in \mathcal{E}_{\mathcal{L}}(\hat{G})$ **do**
5: Let S be the set of positive sibling nodes that have regular sibling paths from a
6: Insert $\langle S, b\rangle$ into $\mathcal{E}_{\mathcal{L}}(\hat{G})$
7: Remove a from the sources of Lambek edges
8: Delete a and all incident edges from $\hat{G}$
9: **return** $\hat{G}$

Proposition 3.4.12. *Given an incrementally integral ATG $\hat{G}$, a PTG G that $\hat{G}$ represents and a positive atom $a \in \mathcal{V}(\hat{G})$, in the span $\mathcal{N}(G)$, ContractPositiveAtom($\hat{G}$, a) returns an ATG $\hat{G}'$ that also represents G.*

Proof. The key insight to this proof is that the edges inserted in algorithm 5 bypass the vertex, which preserves the paths in the ATG.

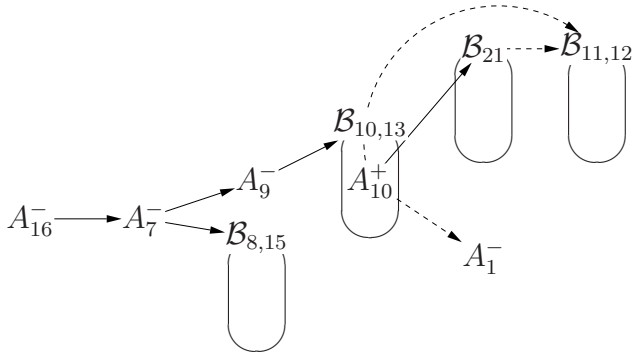
We will prove the clauses of definition 3.4.5 individually.

The first clause of definition 3.4.5 requires that $\mathcal{V}(\hat{G}')$ include the vertices that are peripheral to the span $\mathcal{N}(G)$. Since $\hat{G}$ includes those vertices and the only vertex that ContractPositiveAtom deletes is a, which is not peripheral to $\mathcal{N}(G)$, the first clause of definition 3.4.5 is satisfied.

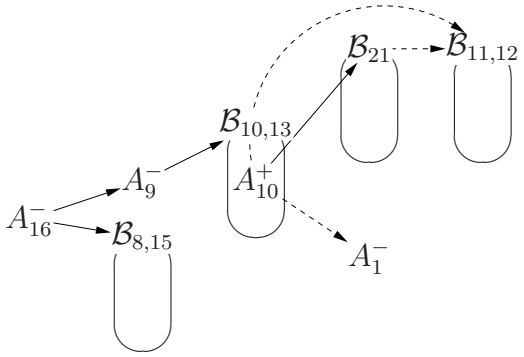
The second clause of definition 3.4.5 requires that each edge in $\mathcal{E}_{\mathcal{R}}(\hat{G}')$ represent some regular path in G. Consider an edge $\langle b, c\rangle \in \mathcal{E}_{\mathcal{R}}(\hat{G})$ inserted at line 3. We know that there exist edges $\langle b, a\rangle$ and $\langle a, c\rangle$ in $\mathcal{E}_{\mathcal{R}}(\hat{G})$. Therefore, there is a regular path in G from b to c and $\langle b, c\rangle$ represents that path.

The third clause of definition 3.4.5 requires that each regular path in G have a representative regular sibling path in $\hat{G}'$. Consider a regular path P in G that does not include a. Then, if P has a representative regular sibling path in $\hat{G}$, it also has a representative regular sibling path in $\hat{G}'$, since a is not on that path. Now, consider a regular path P in G that includes a. If a is either the first or last vertex in P, then the third clause is trivially satisfied. If a is an intermediate vertex in P, then let b be the regular in-neighbour of a in $\hat{G}$, let d be the initial vertex of the regular sibling path in $\hat{G}$ that represents P and let e be its final vertex. Then, P must include one of the component atoms of one of the sibling nodes c in the out-neighbourhood of a in $\hat{G}$. Then, there must be a regular sibling path from d to b to c to e in $\hat{G}'$, because line 3 has added $\langle b, c\rangle$ and this regular sibling path represents P.

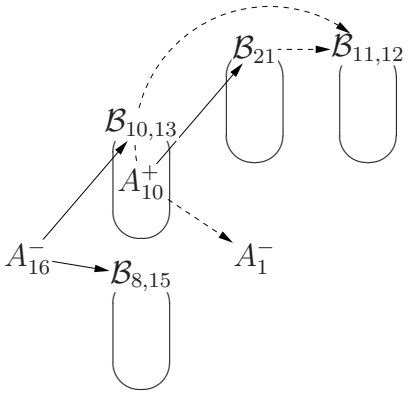
The fourth clause of definition 3.4.5 requires that each Lambek edge in $\hat{G}'$ represent some Lambek edge in G. Lambek edges which are not incident to a and not inserted at line 6 also appear in $\hat{G}$ and since $\hat{G}$ represents G, they represent Lambek edges in G. Let $\langle S, b\rangle$ be a Lambek edge inserted at line 6. Let $\langle a', b'\rangle \in \mathcal{E}_{\mathcal{L}}(G)$ be the edge that $\langle \{a\}, b\rangle$ represents. Then, $\langle S, b\rangle$ represents $\langle a', b'\rangle$, by definition 3.4.3 and the fact that



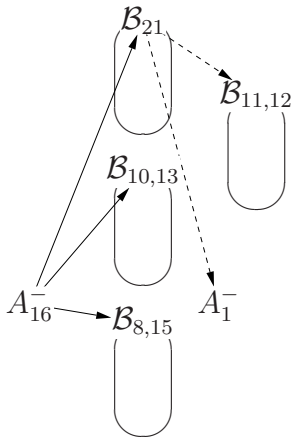
(a) The ATG from figure 3.23f.



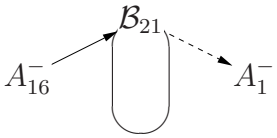
(b) The ATG after 1 vertex contraction.



(c) The ATG after 2 vertex contractions.



(d) The ATG after 3 vertex contractions.



(e) The ATG after vertex contraction.

Figure 3.25: An example of the process for contracting the vertices of a non-concise ATG.

the regular sibling paths in $\hat{G}$ from a to each vertex in S represent regular paths in G. We must also consider the Lambek edges whose sources were modified in line 7. Such Lambek edges need to have replacement edges added whose sources are the vertices with regular sibling paths from a.

The final clause of definition 3.4.5 requires that each Lambek edge in G, for which there is no regular path from its source to its target, has a representative in $\hat{G}'$ if the vertices for representing that edge are in $\mathcal{V}(\hat{G}')$. Consider a Lambek edge in G whose source is a. If such an edge has a representative in $\hat{G}$, then the target of that representative

in $\mathcal{V}(\hat{G})$ must also be in $\mathcal{V}(\hat{G}')$, since the only difference in their vertex sets is a. Therefore, the edge $\langle S, b\rangle$ inserted at line 6 represents that edge in G. □

Algorithm 6 ContractNegativeAtom($\hat{G}$, a)

1: Let b be the regular in-neighbour of a in $\hat{G}$
2: **for** each edge $\langle a, c\rangle \in \mathcal{E}_{\mathcal{R}}(\hat{G})$ **do**
3: Insert the edge $\langle b, c\rangle$ into $\mathcal{E}_{\mathcal{R}}(\hat{G})$
4: **for** each edge $\langle C, \mathcal{B}(a)\rangle \in \mathcal{E}_{\mathcal{L}}(\hat{G})$ **do**
5: Insert the edge $\langle C, b\rangle$ into $\mathcal{E}_{\mathcal{L}}(\hat{G})$
6: **for** each edge $\langle C, a\rangle \in \mathcal{E}_{\mathcal{L}}(\hat{G})$ **do**
7: Insert the edge $\langle C, b\rangle$ into $\mathcal{E}_{\mathcal{L}}(\hat{G})$
8: Delete a and all incident edges from $\hat{G}$
9: **return** $\hat{G}$

Proposition 3.4.13. *Given an incrementally integral ATG $\hat{G}$, a PTG G that $\hat{G}$ represents and a negative atom $a \in \mathcal{V}(\hat{G})$ that is in $\mathcal{N}(G)$, ContractNegativeAtom($\hat{G}$, a) returns an ATG $\hat{G}'$ that also represents G.*

Proof. The key insight to this proof is that the edges inserted in algorithm 6 bypass the vertex, which preserves the paths in the ATG.

The first clause of definition 3.4.5 is proven in the same way as the proof for proposition 3.4.12.

The second clause requires that each edge in $\mathcal{E}_{\mathcal{R}}(\hat{G}')$ represent some regular path in G. First, consider those edges inserted at line 3. Since there is an edge $\langle b, a\rangle \in \mathcal{E}_{\mathcal{R}}(\hat{G})$ and an edge $\langle a, c\rangle \in \mathcal{E}_{\mathcal{R}}(\hat{G})$ and $\hat{G}$ represents G, we know that there is a regular path from b to c in G. Next, consider those edges in $\mathcal{E}_{\mathcal{R}}(\hat{G}')$ that are in $\mathcal{E}_{\mathcal{R}}(\hat{G})$ but are not incident to a. Such edges represent paths in G, since $\hat{G}$ represents G, which implies that the same edges in $\hat{G}'$ represent paths, since they do not include a.

The third clause requires that regular paths in G have representative regular sibling paths in $\hat{G}'$, if the vertices exist to represent them. All regular paths in G that have representatives in $\hat{G}$ trivially have representatives in $\hat{G}'$, except for those including a. Consider a regular path P in G that includes a. If a is the initial or final vertex in P, then the third clause is trivially satisfied. If a is an intermediate node, then let d be the initial vertex of P and let e be its final vertex. Then, P must contain a vertex c in the out-neighbourhood of a in $\hat{G}$. Then, the regular sibling path from d to b to c to e in $\hat{G}'$ represents P

The fourth clause of definition 3.4.5 requires that Lambek edges in $\hat{G}'$ represent some Lambek edge in G. Those Lambek edges in $\hat{G}'$ not inserted at lines 5 and 7 represent the same edges in G that they represent in $\hat{G}$. The Lambek edges inserted at line 5 represent the same edge in G that the edge in line 4 represents, since there is a regular edge $\langle b, a\rangle$ in $\hat{G}$. Similarly, the Lambek edges inserted at line 7 represent the same edge in G as the edge in line 6 represents.

The final clause of definition 3.4.5 requires that Lambek edges in G that could be represented in $\hat{G}'$ are. The only Lambek edges that are deleted from $\hat{G}$ are those whose target is a and they are replaced by Lambek edges in $\hat{G}'$ for which there is a regular sibling path in $\hat{G}$ from what their target represents to what a represents. □

Algorithm 7 ContractSiblingNode($\hat{G}$, b)

1: **if** b is positive **then**
2: Remove b from all sources of Lambek edges in $\hat{G}$
3: Delete b and all incident edges from $\hat{G}$
4: **return** $\hat{G}$

Proposition 3.4.14. *Given an incrementally integral ATG $\hat{G}$, a PTG G that $\hat{G}$ represents and a sibling node $b \in \mathcal{V}(\hat{G})$ that is in the span $\mathcal{N}(G)$ and such that for $a \in b$, $a \notin \mathcal{V}(\hat{G})$, ContractSiblingNode($\hat{G}$, b) returns an ATG $\hat{G}'$ that also represents G.*

Proof. The key insight to this proof is that empty sibling nodes that are not peripheral do not represent any structure that is necessary to determine integrity, and so can be deleted.

First we prove the result for the case where b is positive. Since b is in the span $\mathcal{N}(G)$, we know that each component atom a of b is not in $\mathcal{V}(\hat{G})$. Since $\hat{G}$ represents G, but does not contain any atom occurrences in b, any paths in G including those vertices that could have representatives in $\hat{G}$ must have them without including b since those atom occurrences do not occur in $\hat{G}$. Since the component atoms of b do not appear in $\hat{G}$, deleting b preserves those representatives.

Now, consider the case where b is negative. Again, since no atom $a \in b$ appears in $\hat{G}$, but $\hat{G}$ represents G, any edges with b as a target must already have a representative in $\hat{G}'$. Therefore, deleting b preserves those representatives. □

Proposition 3.4.15. *Given an incrementally integral ATG $\hat{G}$ and a PTG G that $\hat{G}$ represents, repeatedly applying ContractPositiveAtom to the positive atoms of $\hat{G}$, then repeatedly applying ContractNegativeAtom to the negative atoms of $\hat{G}$ and then repeatedly applying ContractSiblingNode to the empty sibling nodes of $\hat{G}$ returns an ATG $\hat{G}'$ that is the concise representative of G.*

Proof. By induction on the applications of the three contraction algorithms, repeatedly applying the contraction algorithms will yield an ATG $\hat{G}'$ that represents G, according to propositions 3.4.12, 3.4.13 and 3.4.14. However, each of ContractPositiveAtom and ContractNegativeAtom delete the atom, which means that all of the atoms in $\hat{G}'$ are peripheral to G. Furthermore, ContractSiblingNode deletes all of the non-peripheral sibling nodes, which proves that $\hat{G}'$ is the concise representative of G. □

Computational Complexity

In this section, we will analyze the time complexity of algorithms 5, 6 and 7.

As in section 3.4.3, we assume that we will only be contracting vertices on ATGs that are the result of either the Bracketing or Adjoining algorithms or one of the vertex contraction algorithms. Such graphs will have a vertex set of size at most $4k$, a regular edge set of size at most $(4k)^2$ and a Lambek edge set of size at most $2^{4k}k$.

In lines 1 through 3 of algorithm 5, we iterate over the vertices in the out-neighbourhood of a, which takes time $O(4k)$. Lines 4 through 6 need to find the set of sibling nodes that have regular sibling paths from a which takes time $O((4k)^2)$ via a simple search. Then, there can be at most $4k$ such vertices each of which requires a constant time operation on line 6. Lines 7 through 9 take time $O(2^{4k}k)$, because they iterate over Lambek edges. Therefore, all of algorithm 5 can be computed in time $O(2^{4k}k)$ time.

Line 2 of algorithm 6 iterates over the regular out-neighbourhood of a in $\hat{G}$, which takes time $O(4k)$. Then, lines 4 and 6 iterate over the vertices in the in-neighbourhood of $\mathcal{B}(a)$, which takes time $O(4k)$. Finally, line 8 takes time $O(^{4k}k)$, because it iterates over edges in the ATG. Therefore, the whole algorithm takes time $O(2^{4k}k)$.

Algorithm 7 is the simplest of the three. Line 1 and 2 iterate over the Lambek edges in $\hat{G}$, so take time $O(2^{4k}k)$. Then, line 3 of that algorithm takes time $O(4k)$, since we need only modify the neighbourhood of B, which takes time proportional to the number of vertices in the graph. Therefore, the whole algorithm takes time $O(2^{4k}k)$.

An important point to note concerning these algorithms is that all of them are proportional only to the maximum order of categories in the proof frame and none are dependant on the number of atom occurrences in the proof frame. This is crucial to our efficiency arguments, since bounding that order by a constant yields constant running times for these algorithms.

3.5 The Parsing Algorithm for L^*

The parsing problem for L^*, or Lambek Categorial Grammar without product and allowing empty premises, is the problem which, given a sentence and a lexicon, asks whether there is an assignment of categories in the lexicon to words in the sentence such that the sequent consisting of those categories is derivable in L^*. The preceding sections introduced increasingly complex mathematical structures for representing proofs in L and L^* and, in this section, we will introduce the chart parsing algorithm that uses those structures to parse L^*.

3.5.1 The Chart

Our chart will be the repository for information concerning incrementally integral PTGs in the form of the ATGs for those PTGs.

The words in the sentence have an inherent order, and the atoms in the proof frame of a category also have an inherent order. The categories in the lexicon for a given word however, do not, and so we must arbitrarily order them to position them in the chart. We do the same for the sentential categories (i.e. those assigned to τ in the lexicon) and we position these after the categories for the last word.

To specify the algorithm, we will need to be able to traverse the words in the sentence, the categories for a word and the atoms for a category. Given a word occurrence w, its category occurrence set $\mathcal{C}(w)$ is the set of category occurrences assigned to it by the lexicon. Given a category occurrence c, its word occurrence $\mathcal{W}(c)$ is the word occurrence w such that $c \in \mathcal{C}(w)$ and its atom occurrence sequence $\mathcal{A}(c)$ is the sequence of atom occurrences appearing in its proof frame. Given an atom occurrence a, its category occurrence $\mathcal{C}(a)$ is the category c such that $a \in \mathcal{A}(c)$.

$$\begin{aligned} \textit{Time} &: \{NP, S\backslash NP/NP\} \\ \textit{flies} &: \{NP, S\backslash NP\} \\ \tau &: \{S, S\backslash NP\} \end{aligned}$$

Figure 3.26: A simple LCG lexicon.

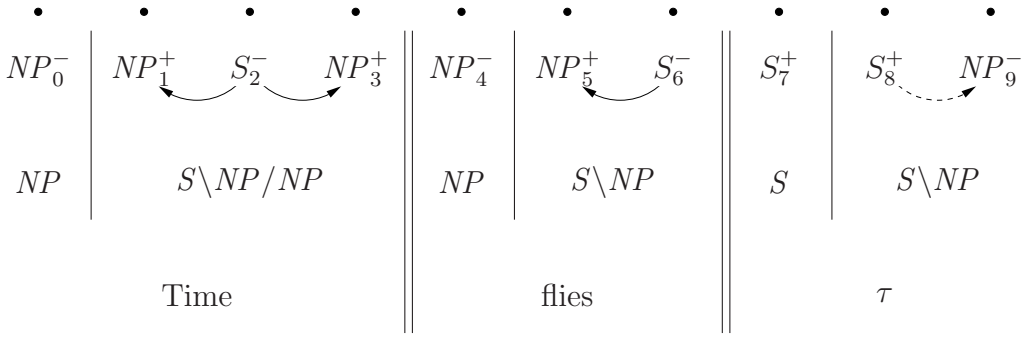


Figure 3.27: The empty chart for the sentence "Time flies" for the lexicon in figure 3.26. τ is the symbol in the lexicon for the sentential categories.

3.5.2 Filling the chart

We will present the algorithm for filling the chart as a number of interrelated methods, which appear as algorithms 8 through 12. As in other chart parsing algorithms, we will insert objects at pairs of indices in the chart. The objects that we will be inserting are ATGs and the indices for our chart will be the atom occurrences of the proof frame.

Given an ATG G over the span $\mathcal{N}(G) = [a_l, a_r]$, G will always be inserted at the pair of indices $\langle a_l, a_r \rangle$. We will distinguish between arcs and entries in the chart in the following way: an *arc* is an ATG together with the pair of atom occurrences at which it is inserted and an *entry* is a pair of atom occurrences together with all of the ATGs inserted at that pair. Or, in other words, an entry is a collection of all of the arcs at a given span.

The required structure to begin the algorithm is shown in figure 3.27. The single vertical lines delineate the boundary between categories for a single word. The double vertical lines delineate the boundary between categories for different words. τ is the symbol in the lexicon that the categories for the sentence are assigned to.

Algorithm 8 outlines the Iterating algorithm. That algorithm takes as input the set of entries, as a 2-dimensional array and indexed by atom occurrences, and the sequence of words in the sentence. This gives the algorithm access to the proof frame via the $\mathcal{A}$ operator, which returns the sequence of atom occurrences for a category, and the $\mathcal{C}$ operator, which returns the sequence of category occurrences for a word. Then, the algorithm iterates over the proof frame and issues calls to two sub-algorithms: AttemptBracketing, which determines whether a certain bracketing is possible, and ExtendArc, which considers a number of potential adjoinings and bracketings.

Algorithm 8 consists of three for loops at the top level. The first for loop iterates over all of the categories for words in the sentence. This loop first calls Bracketing with the empty ATG on each pair of adjacent atoms originating from the same category. Then it calls ExtendArc on all of the ATGs in the spans whose endpoints originate from the same category to add the larger bracketings and the adjoinings for these spans. The second for loop iterates over the links between atoms originating from adjacent words, calling Bracketing with the empty ATG on each pair of those atoms. Finally, the third for loop iterates over all of the spans between atoms not in the same word, calling ExtendArc on each, which adds the larger bracketings and adjoinings.

Before we can describe the algorithm for extending arcs, we must specify a notation for referring to the indices of atom and word occurrences in the proof frame.

Definition 3.5.1. Let F be a proof frame. Given an atom occurrence a, the index of a in F, $\mathcal{I}_{\mathcal{A}}(a)$, is the number of atoms preceding it according to $\mathcal{O}(F)$. Given a word occurrence w, the index of w in F, $\mathcal{I}_{\mathcal{W}}(w)$, is the number of words preceding it in the sentence.

Algorithm 8 Iterating(entries, words)

```
1:  for w ∈ words do
2:      for c ∈ C(w) do
3:          for i ∈ {0, |A(c)| − 1} do
4:              AttemptBracketing(∅, A(c)[i], A(c)[i + 1])
5:          for i ∈ {1, |A(c)| − 1} do
6:              for j ∈ {0, |A(c)| − i − 1} do
7:                  for G ∈ entries[A(c)[i]][A(c)[i + j]] do
8:                      ExtendArc(G, A(c)[i], A(c)[i + j])
9:  for i ∈ {0, |words| − 1} do
10:     for c ∈ C(words[i]) do
11:         for d ∈ C(words[i + 1]) do
12:             AttemptBracketing(∅, A(c)[|A(c)| − 1], A(d)[0])
13: for i ∈ {1, |words| − 1} do
14:     for j ∈ {0, |words| − i − 1} do
15:         for c ∈ C(words[i]) do
16:             for d ∈ C(words[i + j] do
17:                 for a ∈ A(c) do
18:                     for b ∈ A(d) do
19:                         for G ∈ entries[a][b] do
20:                             ExtendArc(G, a, b)
```

Algorithm 9 takes as input an ATG $\hat{G}$ and the two atom occurrences at either end of its span a_1 and a_2. Lines 2 and 3 of that algorithm attempt to bracket $\hat{G}$ in every possible way and the remainder of the algorithm attempts to adjoin $\hat{G}$ in every possible way. The adjoinings are only attempted with ATGs whose spans are either (i) smaller or (ii) of equal size and to the left, where size is defined according to the location of the endpoints of the span of the ATG in section 3.5.3. To do this, the algorithm uses the conditional on line 4. If a_1 and a_2 originate from the same category, the gap between the two is calculated as the difference between the indices of the two atoms on line 5. Then, there are three for loops that consider adjoinings: the one on line 6 that considers smaller adjoinings to the left, the one on line 10 that considers smaller adjoinings to the right and the one on line 14 that considers equal-sized adjoinings to the left. If a_1 and

a_2 do not originate from the same category, the gap between the two is calculated as the difference between the indices of the words of the two atoms, on line 18. Then, the four loops on lines 19 through 35 mirror the for loops on lines 6 through 16.

Algorithm 9 ExtendArc($\hat{G}$, a_1, a_2)

```
 1: Let ⟨P, N⟩ = SurroundingAtoms(a_1, a_2)
 2: for p ∈ P and n ∈ N do
 3:     AttemptBracketing(Ĝ, p, n)
 4: if a_1 and a_2 originate from the same category then
 5:     Let g = I_A(a_2) − I_A(a_1)
 6:     for i ∈ {I_A(a_1) − g, I_A(a_1) − 1} do
 7:         for p ∈ P do
 8:             for Ĥ ∈ entries[A(C(a_1))[i]][p] do
 9:                 AttemptAdjoining(Ĥ, Ĝ)
10:     for i ∈ {I_A(a_2) + 1, I_A(a_2) + g} do
11:         for n ∈ N do
12:             for Ĥ ∈ entries[n][A(C(a_2))[i]] do
13:                 AttemptAdjoining(Ĝ, Ĥ)
14:     for p ∈ P do
15:         for Ĥ ∈ entries[A(C(a_1))[I_A(a_1) − g − 1]][p] do
16:             AttemptAdjoining(Ĥ, Ĝ)
17: else
18:     Let g = I_W(a_2) − I_W(a_1)
19:     for i ∈ {I_W(a_1) − g, I_W(a_1) − 1} do
20:         for c ∈ C(words[i]) do
21:             for a ∈ A(c) do
22:                 for p ∈ P do
23:                     for Ĥ ∈ entries[a][p] do
24:                         AttemptAdjoining(Ĥ, Ĝ)
25:     for i ∈ {I_W(a_2) + 1, I_W(a_2) + g} do
26:         for c ∈ C(words[i]) do
27:             for a ∈ A(c) do
28:                 for n ∈ N do
29:                     for Ĥ ∈ entries[n][a] do
30:                         AttemptAdjoining(Ĝ, Ĥ)
31:     for c ∈ C(words[I_W(a_1) − g − 1]) do
32:         for a ∈ A(c) do
33:             for p ∈ P do
34:                 for Ĥ ∈ entries[a][p] do
35:                     AttemptAdjoining(Ĥ, Ĝ)
```

Algorithm 10 is an algorithm for taking a span $[a_1, a_2]$ and finding the set of atom occurrences that are to the left of a_1 and the set of atoms that are to the right of a_2. In this case, we are not searching for the atom occurrences that precede a_1 and a_2 according to $\mathcal{O}(F)$. Instead, if a_1 is the first atom occurrence in its category occurrence, we are interested in the rightmost atoms in each category occurrence of the word occurrence that precedes the word occurrence of a_1. Otherwise, SurroundingAtoms finds the atom occurrence that precedes a_1 in $\mathcal{O}(F)$. The same notions apply to the atom occurrences following a_2.

Algorithm 10 SurroundingAtoms(a_1, a_2)

if a_1 is the leftmost atom in its category and its word is the first word **then**
  $P = \emptyset$
else if a_1 is the leftmost atom in its category **then**
  $P = \{A(c)[|A(c)| - 1] | c \in C(words[\mathcal{I}_{\mathcal{W}}(W(C(a_1))) - 1])\}$
else
  $P = \{C(a_1)[\mathcal{I}_{\mathcal{A}}(a_1) - 1]\}$
if a_2 is the rightmost atom in its category and its word is the last word **then**
  $N = \emptyset$
else if a_2 is the rightmost atom in its category **then**
  $N = \{A(c)[0] | c \in C(words[\mathcal{I}_{\mathcal{W}}(W(C(a_2))) + 1])\}$
else
  $N = \{C(a_2)[\mathcal{I}_{\mathcal{A}}(a_2) + 1]\}$
return $\langle P, N \rangle$

The final components of the algorithm for filling the chart are the AttemptBracketing and AttemptAdjoining algorithms shown in algorithms 11 and 12. These algorithms are the interface between the iteration algorithms presented in this section and the ATG manipulating algorithms of section 3.4. These algorithms take the output of the Bracketing and Adjoining algorithms from section 3.4 and check its incremental integrity, before building a concise representative and inserting it into the chart, if necessary.

Once the chart has been filled, the final step of the algorithm is to check whether certain spans in the chart contain ATGs. If one of these ATGs exists, then the sentence is grammatical and otherwise the sentence is not grammatical.

Algorithm 11 AttemptBracketing($\hat{G}$, a_1, a_2)

1: Let $\hat{H}$ = Bracketing(G, a_1, a_2)

2: InsertConciseRepresentative($\hat{H}$)

Algorithm 12 AttemptAdjoining($\hat{G}_1$, $\hat{G}_2$)

1: Let $\hat{G}$ = Adjoining($\hat{G}_1$, $\hat{G}_2$)

2: InsertConciseRepresentative($\hat{G}$)

The spans that need to be checked are those spans whose leftmost atom occurrence is the leftmost atom occurrence in a category occurrence for the first word occurrence in the sentence and whose rightmost atom occurrence is the rightmost atom occurrence in a category occurrence for τ.

Figure 3.28 shows a completed chart for the sentence "Time flies". There are two spans from a leftmost category for "Time" to a rightmost category for τ with an empty ATG: the span from NP_1^+ to NP_9^- and the span from NP_0^- to S_7^+. Each such span demonstrates that the sentence is derivable in L^*.

Algorithm 13 InsertConciseRepresentative($\hat{G}$)

```
1: if Ĝ is incrementally integral then
2:     Let A be the subset of 𝒱(Ĝ) that consists of atoms that are not peripheral to Ĝ
3:     for a ∈ A do
4:         if a is positive then
5:             ContractPositiveAtom(Ĝ, a)
6:         else
7:             ContractNegativeAtom(Ĝ, a)
8:     Let B be the subset of 𝒱(Ĝ) that consists of sibling nodes that are not peripheral to Ĝ
9:     for b ∈ B do
10:        ContractSiblingNode(Ĝ, b)
11:    Add Ĝ to entries(a_1, a_2)
```

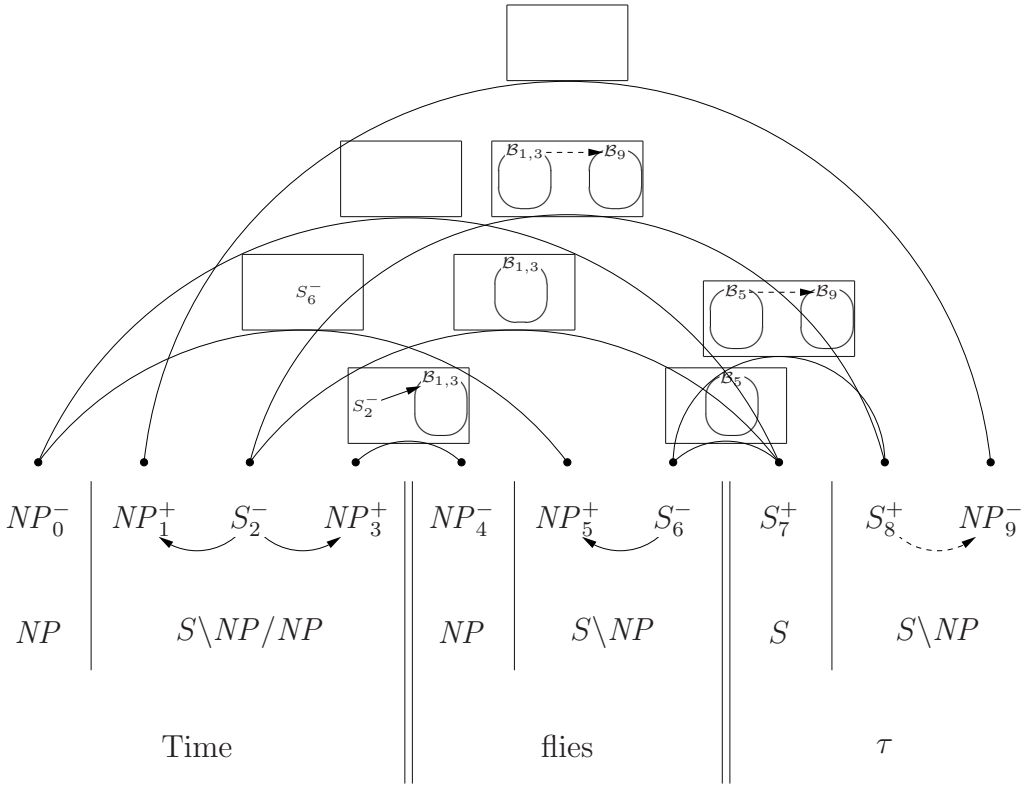


Figure 3.28: The resulting chart for the sentence "Time flies" for the lexicon in figure 3.26.

3.5.3 Correctness

The chart parsing algorithm presented in section 3.5.2 fills in a chart of ATGs, which represent the PTGs over the categories for the words in the sentence. We will provide an intuitive argument for its correctness, based on the preceding proofs and intuitive arguments.

The key insight to this argument is that once we have inserted all of the incrementally integral ATGs into the chart, we need only check whether the ATGs that represent incrementally integral term graphs are present or not.

First, we argue that algorithm 13 inserts a concise representative ATG of a PTG G iff there is some incrementally integral PTG G that $\hat{G}$ represents. Then, algorithm 13 checks that its input is incrementally integral, using algorithms 3.4.12, 3.4.13 and

3.4.14, it obtains a concise representative ATG for any PTGs for which its input is a representative.

Next, we argue that algorithm 10 is sufficient to consider all of the atoms adjacent to a span. This algorithm takes a pair of atom occurrences $\langle a_1, a_2 \rangle$ and returns a pair of sets of atom occurrences $\langle P, N \rangle$ in the proof frame. If a_1 is not the first atom occurrence in its category occurrence, then P is the singleton set consisting of the atom occurrence that immediately precedes a_1 according to $\mathcal{O}(F)$. Otherwise, P is the set of atom occurrences that are the last atom occurrence in the category occurrence for all category occurrences of the word occurrence that immediately precedes the word occurrence for a_1. In both cases, P is the set of atom occurrences that precede a_1 in some sequent for the sentence. Similarly, N is the set of atom occurrences that follows a_2 in some sequent for the sentence.

Before we can proceed to argue for the correctness of algorithm 9, we need to define a notion of the length of a span of a proof frame. The length of a span $[a_l, a_r]$ is a pair $\langle k_w, k_a \rangle$, where k_w is the number of word occurrences between the word occurrence for a_l and the word occurrence for a_r, and k_a is the sum of two values: the number of atom occurrences that are in the category occurrence of a_l but to the right of a_l, and the number of atom occurrences that are in the category occurrence of a_r but to the left of a_r. Then, a span $[a_l, a_r]$ with length $\langle k_w, k_a \rangle$ is smaller than a span $[b_l, b_r]$ with length $\langle l_w, l_a \rangle$ if and only if $k_w < l_w$ or $k_w = l_w$ and $k_a < l_a$. Furthermore, they are of equal size if $k_w = l_w$ and $k_a = l_a$.

We can now argue that algorithm 9 with input $\hat{G}$ considers all bracketings for $\hat{G}$ and all adjoinings that are smaller than $\mathcal{A}(\hat{G})$ or the same size and to the left of $\mathcal{A}(\hat{G})$. By the correctness of algorithm 10, lines 2 and 3 attempt to bracket all of the possible bracketings that $\hat{G}$ could be a part of. Furthermore, lines 4 through 35 iterate over all of the possible adjoinings from smallest to largest that $\hat{G}$ could participate in.

Finally, we can argue that algorithm 8 considers all possible spans in the chart, in

an order such that an ATG is never added to an entry after that entry has already been an input to the ExtendArc algorithm. First, lines 1 and 2 iterate over the category occurrences in the chart. Lines 3 and 4 add all of the ATGs for two adjacent atom occurrences in a category occurrence. Then lines 5 through 7 insert all of the ATGs for spans whose atoms are entirely contained in a single category occurrence. Having inserted all of the ATGs for spans in a single category occurrence, lines 9 through 12 iterate over the category occurrences for adjacent words, while adding the bracketing ATGs with a span whose leftmost atom is the rightmost atom in a category occurrence and whose rightmost atom is the leftmost atom in a category occurrence for the next word occurrence. Finally, lines 13 through 20 iterate over all of the spans that cross a word boundary from smallest to largest and call ExtendArc on each ATG in that entry.

This completes the argument that algorithm 8 fills the chart with the concise representative ATG for exactly the incrementally integral PTGs for the proof frame. Since term graphs are partial term graphs, it follows that the chart also contains the concise representative ATGs for all of the incrementally integral term graphs, which, according to proposition 3.3.5, means it contains the concise representative ATGs for all of the integral term graphs. ATGs for integral term graphs can only appear in entries in the chart consisting of a span whose leftmost atom occurrence is the leftmost atom occurrence in a category occurrence for the first word in the sentence and whose rightmost atom occurrence is the rightmost atom occurrence in a category occurrence for τ. These are exactly the spans that the algorithm checks for integral term graphs.

3.5.4 Computational Complexity

So far in section 3.5, we have introduced a number of algorithms that together fill the chart. We have also introduced a method for checking the ATGs in the chart for those that represent integral term graphs. In this subsection, we will examine the computational complexity of the entire algorithm. As in section 3.5.3, our analysis will be bottom-up.

Algorithm 13 first checks whether its input $\hat{G}$ is incrementally integral. If it is, the algorithm iteratively contracts the vertices that are not peripheral to $\hat{G}$. In section 3.4.3, we established that checking the incremental integrity of an ATG could be done in time $O(2^{8k}k^2)$, and in section 3.4.4, we established that contracting each vertex could be done in time $O(2^{4k}k)$, where k is the bound on the order of categories. Since there can be at most $4k$ such vertices, the vertex contraction takes time $O(2^{4k}k^2)$ which yields a total time for algorithm 13 of $O(2^{8k}k^2)$.

Next we examine the time complexity of the AttemptBracketing and AttemptAdjoining algorithms. These algorithms are simple, and only call Bracketing and Adjoining, respectively, and then call the InsertConciseRepresentative algorithm that we discussed in the preceding paragraph. Section 3.4.2 established that the Bracketing algorithm could be computed in time $O(k2^{2k})$ time, which is dominated by the time for the InsertConciseRepresentative algorithm. Section 3.4.2 also established that the Adjoining algorithm takes time in $O(2^{2k}k^2)$, which is again dominated by the time for the InsertConciseRepresentative algorithm.

Next, we will examine the running time of algorithm 10. This algorithm is simply an analysis of the proof frame to find certain sets of atom occurrences for other pairs of atom occurrences. This algorithm iterates over the category occurrences for words, which takes time at most $O(n)$. Moreover, the sets that it returns are of size $O(n)$, which will be important for analyzing the time complexity of algorithm 9.

Algorithm 9 takes a concise ATG in the chart as input, and then attempts to bracket it and adjoin it with certain other ATGs that are also present in the chart. As we showed in the preceding paragraph, line 1 takes time $O(n)$. Then, lines 2 and 3 iterate over the sets returned by the SurroundingAtoms algorithm, attempting a bracketing for each pair. There are $O(n)$ possible pairs of atoms returned by SurroundingAtoms, and each AttemptBracketing call takes time $O(2^{8k}k^2)$ time, resulting in lines 2 and 3 taking time $O(n2^{8k}k^2)$ total time.

The remainder of algorithm 9 iterates over all of the spans in the chart which can contain an ATG that can be adjoined to $\hat{G}$. Next, the algorithm attempts an adjoining with each ATG in that entry in the chart. We will establish two different time bounds depending on conditions in the lexicon.

If the lexicon contains only a constant number of categories for each word, then there are $O(n)$ entries that can contain ATGs that can be adjoined to $\hat{G}$. Each ATG in an entry must contain the same set of vertices, since they are all concise. This results in as many as $O(2^{2k}2^{2^{2k}k})$ ATGs in each entry. Since each AttemptAdjoining call takes time $O(2^{8k}k^2)$, lines 4 through 35 of algorithm 9 take worst case time $O(n2^{10k}2^{2^{2k}k}k^2)$, if each word in the lexicon is assigned at most a constant number of categories.

However, if the lexicon can contain any number of categories per word, the number of entries that could be adjoined with $\hat{G}$ is $O(n^2)$, rather than $O(n)$. This results in a worst case running time of $O(n^2 2^{10k}2^{2^{2k}k}k^2)$ for this more general case.

Finally, we can examine the running time of algorithm 8. The Iterating algorithm iterates over all of the spans in the chart and attempts all of the bracketings and adjoinings that are appropriate for each span. There are $O(n^2)$ such spans, since n is the number of atom occurrences in the proof frame. For each span, one of AttemptBracketing or ExtendArc is called where the running time of the latter dominates the running time of the former.

In the case where the lexicon is restricted to only a constant number of category assignments per word, this results in a running time for Iterating of $O(n^3 2^{10k}2^{2^{2k}k}k^2)$. In the case of the unrestricted lexicon, the resulting running time is $O(n^4 2^{10k}2^{2^{2k}k}k^2)$.

Both of these results are stated in terms of the number of atom occurrences in the proof frame n, as well as the maximum order of categories allowed in the lexicon k. If the latter is bounded by a constant, the running times become $O(n^3)$ in the case of the lexicon being restricted to only a constant number of category assignments per word or $O(n^4)$ for the unrestricted lexicon.

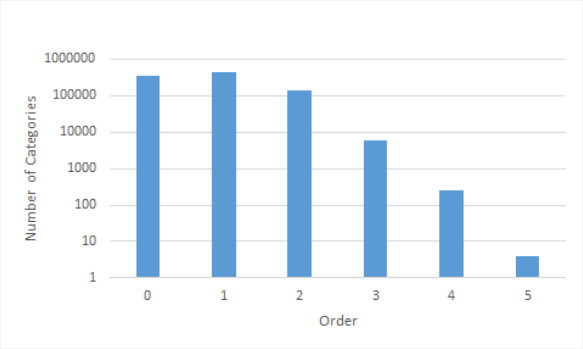


Figure 3.29: The order of categories in CCGbank in sections 02-21.

3.6 Order and CCGbank

The complexity of the algorithm given above is polynomial when the order of categories in the input is bounded by a constant, but its running time is exponential in the order of those categories.

Figure 3.29 shows the distribution of order across the categories occurring in sections 02 to 21 of CCGbank. The maximum order of any category is 5 and the average is 0.785. These results indicate that our parsing algorithm will be efficient, if not competitive, with other categorial grammar parsing strategies, on categories resembling those in CCGbank.

Chapter 4

Practical parsing and non-context-free languages

One of the primary differences between grammar formalisms that are discussed in the literature is the difference in generative capacity. The simplest and most common generative capacity is *weak generative capacity*, which describes the class of string languages that a grammar formalism generates. The weak generative capacity of categorial grammars is generally well-understood. The Combinatory Categorial Grammar (CCG) definition of Vijay-Shanker and Weir (1990) is known to have weak generative capacity greater than Context-Free Grammars (CFGs) and equivalent to Tree-Adjoining Grammars (Vijay-Shanker and Weir, 1994). In contrast, the weak generative capacity of Lambek Categorial Grammar (LCG) is known to be the context-free languages (Pentus, 1997). These generative capacity results, along with the arguments for the non-context-freeness of natural language (Shieber, 1985, Culy, 1985) have caused interest in LCG to wane.

In chapter 2, we saw that CCGbank (Hockenmaier and Steedman, 2007), and as a consequence, the Clark and Curran parser (Clark and Curran, 2007b), deviates from the principles of categorial grammar in many of its derivations. These deviations are subtle, but have important consequences for the generative capacity of the grammar formalism.

In this chapter, we will provide a generalized framework for CCG within which the full variation of CCG seen in the literature can be defined, including the CCG of the Clark and Curran parser. Then, we will prove that the CCG of the Clark and Curran parser generates only the context-free languages. Since the Clark and Curran parser is the state of the art in categorial grammar parsing, these results imply that the weak generative capacity of LCG is not currently a hindrance for practical parsing.

The string language that a grammar formalism can generate is important because it gives some indication of its power, but it ignores any structure beyond strings. Since our primary interest in parsers is in syntax and semantics, we must consider the *strong generative capacity* of a grammar formalism, or rather, the languages of structure that it generates. Our proofs generalize from the case of string languages to the case of derivation languages, proving that the Clark and Curran parser can only generate the structures that a CFG can. This means that we can use standard CFG parsers to parse with CCG, which we will investigate in section 4.2. In particular, we use the Petrov parser (Petrov and Klein, 2007), a probabilistic CFG parser that uses latent variables to refine the grammar extracted from a corpus to improve accuracy, whose original purpose was to improve parsing results on the Penn Treebank. We train the Petrov parser on CCGbank and achieve results that are competitive with the state of the art (Clark and Curran, 2007b, Auli and Lopez, 2011b,a, Xu et al., 2014, 2015) on sentences from the standard test section of CCGbank in terms of supertagging accuracy, PARSEVAL measures and dependency accuracy.

These results should not be interpreted as proof that grammars extracted from the Penn Treebank and from CCGbank are equivalent. Bos's system (Bos et al., 2004) for building semantic terms from CCG derivations is only possible due to the categorial nature of CCG. Furthermore, certain relations between words involved in the linguistic constructions of extraction and coordination have a more natural representation in categorial grammars than in phrase-structure grammars.

4.1 The Language Classes of Combinatory Categorial Grammars

According to the literature, CCG has been defined to have a variety of rule systems. These rule systems vary from a small rule set that is motivated mathematically (Vijay-Shanker and Weir, 1994) to a larger rule set that is motivated linguistically, (Steedman, 2000) to a very large rule set that is motivated by practical coverage (Hockenmaier and Steedman, 2007, Clark and Curran, 2007b). We provide a definition general enough to incorporate these main variants of CCG, as well as others. We now proceed to generalize the definition of CCG from section 2.4.

A CCG is a categorial grammar whose rule system consists of rule schemata, where the left side is a sequence of categories and the right side is a single category, and the categories may include variables over both categories and connectives. In addition, rule schemata may specify a sequence of categories and connectives using the ... convention[1]. When ... appears in a rule, it matches any sequence of categories and connectives according to the connectives adjacent to the .... For example, the rule schema for forward composition is:

$$X/Y, Y/Z \rightarrow X/Z$$

and the rule schema for generalized forward crossed composition is:

$$X/Y, Y|_1 Z_1|_2 \ldots|_n Z_n \rightarrow X|_1 Z_1|_2 \ldots|_n Z_n$$

where X, Y and Z_i for $1 \leq i \leq n$ are variables over categories and $|_i$ for $1 \leq i \leq n$ are variables over connectives (which generally include at least the connectives / and \). Figure 4.1 shows a CCG derivation for the sentence "Mr. Vinken is chairman of Elsevier N.V. , the Dutch publishing group ." from CCGbank that we will refer to throughout

[1]The ... convention (Vijay-Shanker and Weir, 1994) is essentially identical to the $ convention of Steedman (2000).

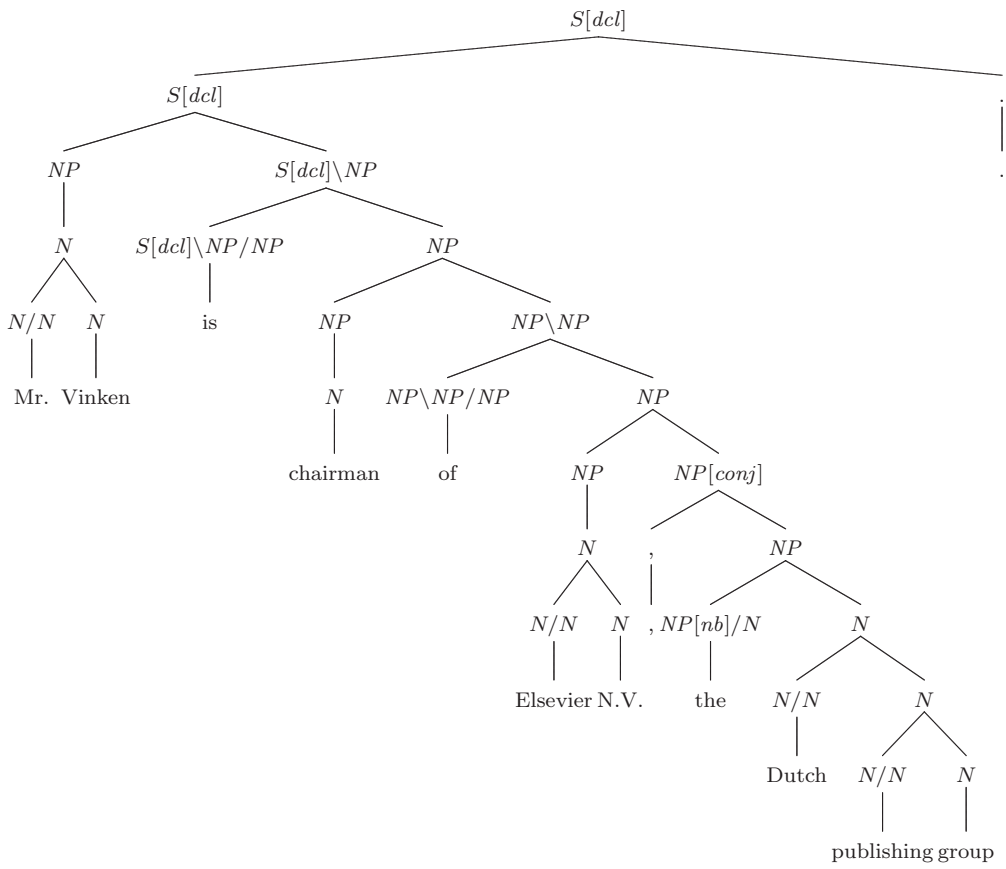


Figure 4.1: The CCGbank phrase-structure tree for sentence 0001.2.

this chapter[2].

4.1.1 Classes for defining CCG

We define a number of *schema classes* general enough that the important variants of CCG can be defined by selecting some subset of the classes. In addition to the schema classes, we also define two *restriction classes* that define ways in which the rule schemata from the schema classes can be restricted.

[2]CCGbank sentence 0001.2.

Schema Classes

(1) Application

- $X/Y, Y \to X$
- $Y, X\backslash Y \to X$

(2) Composition

- $X/Y, Y/Z \to X/Z$
- $Y\backslash Z, X\backslash Y \to X\backslash Z$

(3) Crossed Composition

- $X/Y, Y\backslash Z \to X\backslash Z$
- $Y/Z, X\backslash Y \to X/Z$

(4) Generalized Composition

- $X/Y, Y/Z_1/\ldots/Z_n \to X/Z_1/\ldots/Z_n$
- $Y\backslash Z_1\backslash\ldots\backslash Z_n, X\backslash Y \to X\backslash Z_1\backslash\ldots\backslash Z_n$

(5) Generalized Crossed Composition

- $X/Y, Y|_1 Z_1|_2 \ldots|_n Z_n \to X|_1 Z_1|_2 \ldots|_n Z_n$
- $Y|_1 Z_1|_2 \ldots|_n Z_n, X\backslash Y \to X|_1 Z_1|_2 \ldots|_n Z_n$

(6) Reducing Generalized Crossed Composition

Generalized Composition or Generalized Crossed Composition where $|X| \leq |Y|$.

(7) Substitution

- $(X/Y)|_1 Z, Y|_1 Z \to X|_1 Z$
- $Y|_1 Z, (X\backslash Y)|_1 Z \to X|_1 Z$

(8) D Combinator[3]

- $X/(Y|_1Z), Y|_2W \rightarrow X|_2(W|_1Z)$
- $Y|_2W, X\backslash(Y|_1Z) \rightarrow X|_2(W|_1Z)$

(9) Type-Raising

- $X \rightarrow T/(T\backslash X)$
- $X \rightarrow T\backslash(T/X)$

(10) Finitely Restricted Type-Raising

- $X \rightarrow T/(T\backslash X)$ where $\langle X, T\rangle \in S$ for finite S
- $X \rightarrow T\backslash(T/X)$ where $\langle X, T\rangle \in S$ for finite S

(11) Finite Unrestricted Variable-Free Rules

- $\vec{X} \rightarrow Y$ where $\langle \vec{X}, Y\rangle \in S$ for finite S

Restriction Classes

(A) Rule Restriction to a Finite Set

The rule schemata in the schema classes of a CCG are limited to a finite number of instantiations.

(B) Rule Restrictions to Certain Categories[4]

The rule schemata in the schema classes of a CCG are limited to a finite number of instantiations, although variables are allowed in the instantiations.

[3]Hoyt and Baldridge (2008) argue for the inclusion of the D Combinator in CCG.

[4]Baldridge (2002) introduced a variant of CCG, where modalities are added to the connectives / and \, along with variants of the combinatory rules based on these modalities. Our proofs about restriction class (B) are essentially identical to proofs regarding the multi-modal variant.

The first two definitions of CCG fit directly from the literature into this framework. Vijay-Shanker and Weir (1994) define CCG to be schema class (4) with restriction class (B). Steedman (2000) defines CCG to be schema classes (1–5), (7), (10) with restriction class (B).

4.1.2 Strongly Context-Free CCGs

We will now proceed to identify precisely which CCGs within this framework generate non-context-free languages.

Proposition 4.1.1. *The set of atoms in any derivation of any CCG consisting of a subset of the schema classes (1–8) and (10–11) is finite.*

Proof. A finite lexicon can introduce only a finite number of atoms in lexical categories.

Any rule corresponding to a schema in the schema classes (1–8) has only those atoms on the right that occur somewhere on the left. Rules in classes (10–11) can each introduce a finite number of atoms, but there can be only a finite number of such rules, limiting the new atoms to a finite number. □

Definition 4.1.2. The *subcategories* for a category c are c_1 and c_2 if $c = c_1 \bullet c_2$ for $\bullet \in \{/, \backslash\}$ and c if c is atomic. Its *second subcategories* are the subcategories of its subcategories.

Proposition 4.1.3. *Any CCG consisting of a subset of the schema classes (1–3), (6–8), and (10–11) has derivations consisting of only a finite number of categories.*

Proof. We first prove the proposition excluding schema class (8). We will use structural induction on the derivations to prove that there is a bound on the size of the subcategories of any category in the derivation. The base case is the assignment of a lexical category to a word, and the inductive step is the use of a rule from schema classes (1–4), (6–7) and (10–11).

Given that the lexicon is finite, there is a bound k on the size of the subcategories of lexical categories. Furthermore, there is a bound l on the size of the subcategories of categories on the right side of any rule in (10) and (11). Let $m = max(k, l)$.

For rules from schema class (1), the category on the right is a subcategory of the first category on the left, so the subcategories on the right are bounded by m. For rules from schema classes (2–3), the category on the right has subcategories X and Z, each of which is bounded in size by m since they occur as subcategories of categories on the left.

For rules from schema class (6), since reducing generalized composition is a special case of reducing generalized crossing composition, we need only consider the latter. The category on the right has subcategories $X|_1Z_1|_2 \ldots |_{n-1}|Z_{n-1}$ and Z_n. Z_n is bounded in size by m because it occurs as a subcategory of the second category on the left. Then, the size of $Y|_1Z_1|_2 \ldots |_{n-1}|Z_{n-1}$ must be bounded by m and since $|X| \leq |Y|$, the size of $X|_1Z_1|_2 \ldots |_{n-1}|Z_{n-1}$ must also be bounded by m.

For rules from schema class (7), the category on the right has subcategories X and Z. The size of Z is bounded by m because it is a subcategory of a category on the left. The size of X is bounded by m because it is a second subcategory of a category on the left.

Finally, the use of rules in schema classes (10–11) have categories on the right that are bounded by l, which are, in turn, bounded by m. Then, by proposition 4.1.1, there must only be a finite number of categories in any derivation in a CCG consisting of a subset of schema classes (1–3), (6-7) and (10–11).

The proof including schema class (8) is essentially identical except that k must be defined in terms of the size of the second subcategories. □

Definition 4.1.4. A grammar is *strongly context-free* if there exists a CFG such that the derivations of the two grammars are identical.

Proposition 4.1.5. *Any CCG consisting of a subset of the schema classes (1–3), (6–8) and (10–11) is strongly context-free.*

Proof. Since the CCG generates derivations whose categories are finite in number, let C be that set of categories. We use the notation $S(C, X)$ to indicate the subset of C matching category X (where X may have variables). Then, for each rule schema $C_1, C_2 \to C_3$ in (1–3) and (6–8), we construct a context-free rule $C_3' \to C_1', C_2'$ for each C_i' in $S(C, C_i)$ for $1 \leq i \leq 3$. Similarly, for each rule schema $C_1 \to C_2$ in (10), we construct a context-free rule $C_2' \to C_1'$ which results in a finite number of such rules. Finally, for each rule schema $\vec{X} \to Z$ in (11) we construct a context-free rule $Z \to \vec{X}$. Then, for each entry in the lexicon $w \to C$, we construct a context-free rule $C \to w$.

The constructed CFG has precisely the same rules as the CCG restricted to the categories in C, except that the left and right sides have been reversed. Thus, by proposition 4.1.3, the CFG has exactly the same derivations as the CCG. □

Proposition 4.1.6. *Any CCG consisting of a subset of the schema classes (1–3), (6–8) and (10–11) along with restriction class (B) is strongly context-free.*

Proof. If a CCG is allowed to restrict the use of its rules to certain categories as in restriction class (B), then we construct the context-free rules by enumerating only those categories in the set C allowed by the restriction. □

Proposition 4.1.7. *Any CCG that includes restriction class (A) is strongly context-free.*

Proof. We construct a CFG with exactly those rules in the finite set of instantiations of the CCG rule schemata, along with context-free rules corresponding to the lexicon. This CFG generates exactly the same derivations as the CCG. □

We have thus proved that a wide range of the schema classes used to define CCGs is context-free.

4.1.3 CCG Definitions in Practice

CCGbank (Hockenmaier and Steedman, 2007) is a corpus of CCG derivations that was semi-automatically converted from the Wall Street Journal section of the Penn Treebank.

	Schema Class	*Rules*	*Instances*
(1)	Application	519	902,176
(2)	Composition	102	7,189
(3)	Crossed Composition	64	14,114
(4)	Generalized Composition	0	0
(5)	Generalized Crossed Composition	0	0
(6)	Reducing Generalized Crossed Composition	50	612
(7)	Substitution	3	4
(9)	Type-Raising	27	3,996
(11)	Unrestricted Rules	642	335,011
	Total	1,407	1,263,102

Table 4.1: The rules of CCGbank by schema class.

Table 4.1 shows a categorization of the rules used in CCGbank according to the schema classes defined in the preceding section, where a rule is placed into the least general class to which it belongs. In addition to having no generalized composition other than the reducing variant, it should also be noted that in all generalized composition rules, $X = Y$. This implies that the reducing class of generalized composition is a very natural schema class for CCGbank.

If we assume that type-raising is restricted to those instances occurring in CCGbank[5], then a CCG consisting of schema classes (1–3), (6–7) and (10–11) can generate all the derivations in CCGbank. By proposition 4.1.5, such a CCG is strongly context-free[6].

[5]Without such an assumption, parsing is intractable.

[6]One could make the argument that since CCGbank is finite, it is context-free simply because it is

The Clark and Curran CCG Parser (Clark and Curran, 2007b) is a CCG parser that uses CCGbank as a training corpus. Despite the fact that there is a strongly context-free CCG that generates all of the derivations in CCGbank, it is still possible that the grammar learned by the Clark and Curran parser is not a Context-Free Grammar. However, in addition to schema classes (1–6) and (10–11), Clark and Curran also include restriction class (A) by restricting rules to only those found in the training data[7]. Thus, by proposition 4.1.7, the Clark and Curran parser is a context-free parser.

This result immediately yields an additional result that we will not prove here formally. Because the Clark and Curran parser uses the CYK parsing algorithm of Steedman (2000), restricting the input grammars to only the subset above that are strongly equivalent to a CFG yields a worst-case parsing time of $O(n^5)$.

4.2 A Latent Variable CCG Parser

The context-freeness of a number of CCGs should not be considered evidence that there is no advantage to CCG as a grammar formalism. Unlike the CFGs extracted from the Penn Treebank, CCG allows for the categorial semantics that accompanies any categorial parse and, also, for a more elegant analysis of linguistic structures, such as extraction and coordination. However, because we now know that the CCG defined by CCGbank is strongly context-free, we can use tools from the CFG parsing community to improve CCG parsing.

To illustrate this point, we train the Petrov parser (Petrov and Klein, 2007) on CCGbank. The Petrov parser uses latent variables to refine a coarse-grained grammar extracted from a training corpus to a grammar that makes much more fine-grained syntactic

a finite object. However, our statement is much stronger than this because this CCG is a very natural grammar to define based on CCGbank and it generates derivations not found in CCGbank but it is also context-free.

[7]The Clark and Curran parser has an option, which is disabled by default, for not restricting the rules to those that appear in the training data. However, they find that this restriction is "detrimental to neither parser accuracy or coverage" (Clark and Curran, 2007b).

distinctions. For example, in Petrov's experiments on the Penn Treebank, the syntactic category NP was refined to the more fine-grained NP^1 and NP^2, which roughly correspond to NPs in subject and object positions. Rather than requiring such distinctions to be made in the corpus, the Petrov parser hypothesizes these splits automatically.

The Petrov parser operates by performing a fixed number of iterations of splitting, merging, and smoothing. The splitting process is done by performing Expectation-Maximization to determine a likely potential split for each syntactic category. Then, during the merging process, some of the splits are undone to reduce grammar size and avoid overfitting according to the likelihood of the split against the training data. Smoothing is then done to eliminate noise in the data.

The Petrov parser was chosen for our experiments because it refines the grammar in a mathematically principled way, without altering the nature of the derivations that are output. This is important because the input to the semantic backend and the system that converts CCG derivations to dependencies requires CCG derivations as they appear in CCGbank.

4.2.1 Experiments

Our experiments use CCGbank as the corpus, and we use sections 02–21 for training (39,603 sentences), section 00 for development (1913 sentences) and section 23 for testing (2407 sentences).

CCGbank, in addition to the basic atoms S, N, NP and PP, also differentiates both the S and NP atoms with *features*, thereby allowing for more subtle distinctions. For example, declarative sentences are $S[dcl]$, wh-questions are $S[wq]$ and sentence fragments are $S[frg]$ (Hockenmaier and Steedman, 2007). These features allow finer control of the use of combinatory rules in the resulting grammars. However, this fine-grained control is exactly what the Petrov parser does automatically. Therefore, we trained the Petrov parser twice, once on the original version of CCGbank (denoted "Petrov" in the tables),

and once on a version of CCGbank without these features (denoted "Petrov no feats" in the tables). Furthermore, we evaluated the parsers obtained after 0, 4, 5 and 6 training iterations (denoted I-0, I-4, I-5 and I-6 in the tables). When we evaluated on sets of sentences where not all parsers returned an analysis, we report the coverage.

We use the `evalb` package for PARSEVAL evaluation, and a modified version of Clark and Curran's `evaluate` script for dependency evaluation. To determine statistical significance, we obtain p-values from Bikel's randomized parsing evaluation comparator[8], modified for use with tagging accuracy, F-score, and dependency accuracy.

Bikel's randomized parsing evaluation employs a type of "stratified shuffling", which is a testing method for pairs of sets of results, which tests whether there is a statistically significant difference between the two sets of results. This method randomly exchanges some subset of the results, and then determines whether this exchange results in a change in the test values. After a large number of such exchanges, the total number that resulted in changed test values is counted, and the p-value is calculated by dividing the total number of changed test results by the total number of exchanges.

4.2.2 Supertag Evaluation

Before evaluating the parse trees that the parsers output as a whole, we evaluate the categories assigned to words. In the supertagging literature, POS tagging and supertagging are distinguished: POS tags are the traditional Penn Treebank tags (e.g. NN, VBZ and DT) and supertags are CCG categories. However, because the Petrov parser trained on CCGbank has no notion of Penn Treebank POS tags, we can only evaluate the accuracy of the supertags.

The results are shown in tables 4.2 and 4.3, where the "Accuracy" column shows accuracy of the supertags against the CCGbank categories, and the "No feats" column shows accuracy when features are ignored. Despite the lack of POS tags in the Petrov

[8]http://www.cis.upenn.edu/~dbikel/software.html

Parser	*Accuracy %*	*No feats %*
Clark and Curran Normal-Form	92.92	93.38
Clark and Curran Hybrid	93.06	93.52
Petrov I-5	**93.18**	93.73
Petrov no feats I-6	-	**93.74**

Table 4.2: Supertagging accuracy on the sentences in section 00 that receive derivations from the four parsers shown.

Parser	*Accuracy %*	*No feats %*
Clark and Curran Hybrid	92.98	93.43
Petrov I-5	**93.10**	93.59
Petrov no feats I-6	-	**93.62**

Table 4.3: Supertagging accuracy on the sentences in section 23 that receive derivations from the three parsers shown.

parser, we can see that it performs slightly better than the Clark and Curran parser. The difference in accuracy is only statistically significant between Clark and Curran's Normal-Form model ignoring features and the Petrov parser trained on CCGbank without features ($p = 0.013$).

4.2.3 Constituent Evaluation

In this section, we evaluate the parsers using the traditional PARSEVAL measures, which measure recall, precision, and F-score on constituents in both labelled and unlabelled versions. In addition, we report a variant of the labelled PARSEVAL measures, where we ignore the features on the categories. We report the PARSEVAL measures for all

	Labelled %			*Labelled no feats %*			
Parser	*R*	*P*	*F*	*R*	*P*	*F*	*Coverage*
Clark and Curran Normal-Form	71.14	70.76	70.95	80.66	80.24	80.45	98.95
Clark and Curran Hybrid	50.08	49.47	49.77	58.13	57.43	57.78	98.95
Petrov I-0	74.19	74.27	74.23	74.66	74.74	74.70	99.95
Petrov I-4	85.86	85.78	85.82	86.36	86.29	86.32	99.90
Petrov I-5	**86.30**	**86.16**	**86.23**	86.84	86.70	86.77	99.90
Petrov I-6	85.95	85.68	85.81	86.51	86.23	86.37	99.22
Petrov no feats I-0	-	-	-	72.16	72.59	72.37	99.95
Petrov no feats I-5	-	-	-	86.67	86.57	86.62	99.90
Petrov no feats I-6	-	-	-	**87.45**	**87.37**	**87.41**	99.84

Table 4.4: Labelled constituent accuracy on all sentences from section 00.

	Unlabelled %			
	R	*P*	*F*	*Coverage*
Clark and Curran Normal-Form	86.16	85.71	85.94	98.95
Clark and Curran Hybrid	61.27	60.53	60.90	98.95
Petrov I-0	78.65	78.73	78.69	99.95
Petrov I-4	89.96	89.88	89.92	99.90
Petrov I-5	90.28	90.13	90.21	99.90
Petrov I-6	90.22	89.93	90.08	99.22
Petrov no feats I-0	76.52	76.97	76.74	99.95
Petrov no feats I-5	90.30	90.20	90.25	99.90
Petrov no feats I-6	**90.99**	**90.91**	**90.95**	99.84

Table 4.5: Unlabelled constituent accuracy on all sentences from section 00.

	Labelled %			*Labelled no feats %*		
Parser	*R*	*P*	*F*	*R*	*P*	*F*
Petrov I-5	86.56	86.46	86.51	87.10	87.01	87.05
Petrov no feats I-6	-	-	-	**87.45**	**87.37**	**87.41**
p	-	-	-	0.089	0.090	0.088

Table 4.6: Labelled constituent accuracy on the sentences in section 00 that receive a derivation from both parsers.

	Unlabelled %		
Parser	*R*	*P*	*F*
Petrov I-5	90.43	90.33	90.38
Petrov no feats I-6	**90.99**	**90.91**	**90.95**
p	0.006	0.008	0.007

Table 4.7: Unlabelled constituent accuracy on the sentences in section 00 that receive a derivation from both parsers.

	Labelled %			*Labelled no feats %*			
Parser	*R*	*P*	*F*	*R*	*P*	*F*	*Coverage*
Clark and Curran Normal-Form	71.15	70.79	70.97	80.73	80.32	80.53	99.58
Petrov I-5	**86.94**	**86.80**	**86.87**	87.47	87.32	87.39	99.83
Petrov no feats I-6	-	-	-	**87.49**	**87.49**	**87.49**	99.96

Table 4.8: Labelled constituent accuracy on all sentences from section 23.

	Unlabelled %			
Parser	*R*	*P*	*F*	*Coverage*
Clark and Curran Normal-Form	86.31	85.88	86.10	99.58
Petrov I-5	90.75	90.59	90.67	99.83
Petrov no feats I-6	**90.81**	**90.82**	**90.81**	99.96

Table 4.9: Unlabelled constituent accuracy on all sentences from section 23.

	Labelled %			*Labelled no feats %*		
Parser	*R*	*P*	*F*	*R*	*P*	*F*
Petrov I-5	86.94	86.80	86.87	87.47	87.32	87.39
Petrov no feats I-6	-	-	-	**87.48**	**87.49**	**87.49**
p	-	-	-	0.463	0.215	0.327

Table 4.10: Labelled constituent accuracy on the sentences in section 23 that receive a derivation from both parsers.

	Unlabelled %		
Parser	*R*	*P*	*F*
Petrov I-5	90.75	90.59	90.67
Petrov no feats I-6	**90.81**	**90.82**	**90.81**
p	0.364	0.122	0.222

Table 4.11: Unlabelled constituent accuracy on the sentences in section 23 that receive a derivation from both parsers.

	Labelled %			*Unlabelled %*			
Parser	*R*	*P*	F	*R*	*P*	*F*	*Coverage*
Petrov on Penn Treebank I-6	89.65	89.97	89.81	90.80	91.13	90.96	100.00
Petrov on CCGbank I-5	86.94	86.80	86.87	90.75	90.59	90.67	99.83
Petrov on CCGbank no feats I-6	87.49	87.49	87.49	90.81	90.82	90.81	99.96

Table 4.12: Constituent accuracy for the Petrov parser on the corpora on all sentences from section 23.

sentences in sections 00 and 23 of CCGbank, rather than for sentences of length less than 40 or less than 100. The results are essentially identical for those two sets of sentences.

Tables 4.4 and 4.5 give the PARSEVAL measures on section 00 for Clark and Curran's two best models, the Petrov parser trained on the original CCGbank, and the version of CCGbank without features, after various numbers of training iterations. Tables 4.8 and 4.9 give the accuracies on section 23.

In the case of Clark and Curran's hybrid model, the poor accuracy relative to the Petrov parsers can be attributed to the fact that this model chooses derivations based on the associated dependencies at the expense of constituent accuracy (see section 4.2.4). In the case of Clark and Curran's normal-form model, the large difference between labelled and unlabelled accuracy is primarily due to the mislabelling of a small number of features (specifically, $NP[nb]$ and $NP[num]$). The labelled accuracies without features gives the results when all features are disregarded.

Due to the similarity of the accuracies and the difference in the coverage between I-5 of the Petrov parser on CCGbank and I-6 of the Petrov parser on CCGbank without features, we reevaluate their results on only those sentences for which they both return

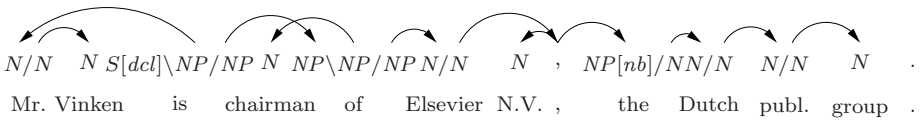


Figure 4.2: The function argument relations for the CCG derivation in figure 4.1.

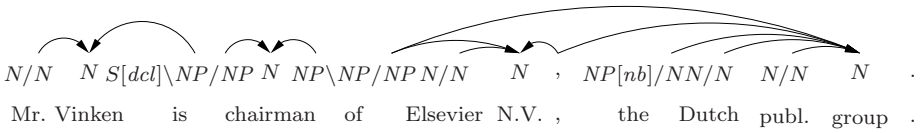


Figure 4.3: The set of dependencies obtained by reorienting the function argument edges in figure 4.2.

derivations in tables 4.6, 4.7, 4.10, and 4.11. These results show that the features in CCGbank actually inhibit accuracy (to a statistically significant degree in the case of unlabelled accuracy on section 00) when used as training data for the Petrov parser.

Table 4.12 gives a comparison between the Petrov parser trained on the Penn Treebank and on CCGbank. These numbers should not be directly compared, but the similarity of the unlabelled measures indicates that the difference between the *structure* of the Penn Treebank and CCGbank is not large.[9]

4.2.4 Dependency Evaluation

The constituent-based PARSEVAL measures are simple to calculate from the output of the Petrov parser, but the relationship of the PARSEVAL scores to the quality of a parse is not entirely clear. In fact, Clark and Curran (2007b) completely ignore PARSEVAL accuracy in favour of dependency accuracy. A dependency evaluation is aided by the fact that in addition to the CCG derivation for each sentence, CCGbank also includes

[9]Because punctuation in CCG can have grammatical function, we include it in our accuracy calculations resulting in lower scores for the Petrov parser trained on the Penn Treebank than those reported by Petrov and Klein (2007).

	Labelled %			*Unlabelled %*			
Parser	*R*	*P*	F	*R*	*P*	*F*	*Coverage*
Clark and Curran Normal-Form	84.39	85.28	84.83	90.93	91.89	91.41	98.95
Clark and Curran Hybrid	84.53	**86.20**	85.36	90.84	**92.63**	91.73	98.95
Petrov I-0	79.87	78.81	79.34	87.68	86.53	87.10	96.45
Petrov I-4	84.76	85.27	85.02	91.69	92.25	91.97	96.81
Petrov I-5	**85.30**	85.87	**85.58**	**92.00**	92.61	**92.31**	96.65
Petrov I-6	84.86	85.46	85.16	91.79	92.44	92.11	96.65

Table 4.13: Dependency accuracy on CCGbank dependencies on all sentences from section 00.

	Labelled %			*Unlabelled %*		
Parser	*R*	*P*	F	*R*	*P*	*F*
Clark and Curran Hybrid	84.71	**86.35**	85.52	90.96	92.72	91.83
Petrov I-5	**85.50**	86.08	**85.79**	**92.12**	**92.75**	**92.44**
p	0.005	0.189	0.187	< 0.001	0.437	0.001

Table 4.14: Dependency accuracy on the section 00 sentences that receive an analysis from both parsers.

	Labelled %			*Unlabelled %*		
Parser	*R*	*P*	F	*R*	*P*	*F*
Clark and Curran Hybrid	85.11	**86.46**	85.78	91.15	92.60	91.87
Petrov I-5	**85.73**	86.29	**86.01**	**92.04**	**92.64**	**92.34**
p	0.013	0.278	0.197	< 0.001	0.404	0.005

Table 4.15: Dependency accuracy on the section 23 sentences that receive an analysis from both parsers.

Parser	*Training Time (CPU minutes)*	*Parsing Time (CPU minutes)*	*Training RAM (gigabytes)*
Clark and Curran Normal-Form	1,152	2	28
Clark and Curran Hybrid	2,672	4	37
Petrov on Penn Treebank I-0	1	5	2
Petrov on Penn Treebank I-5	180	20	8
Petrov on Penn Treebank I-6	660	21	16
Petrov on CCGbank I-0	1	5	2
Petrov on CCGbank I-4	103	70	8
Petrov on CCGbank I-5	410	600	14
Petrov on CCGbank I-6	2,760	2,880	24
Petrov on CCGbank no feats I-0	1	5	2
Petrov on CCGbank no feats I-5	360	240	7
Petrov on CCGbank no feats I-6	1,980	390	13

Table 4.16: Time and space usage when training on sections 02–21 and parsing on section 00.

a set of dependencies. Furthermore, extracting dependencies from a CCG derivation is theoretically well-established (Clark et al., 2002), and bears a strong resemblance to building a proof net from an LCG proof.

A CCG derivation can be converted into dependencies by, first, determining which arguments match which functions, as specified by the CCG derivation. This can be represented as in figure 4.2. Although this is not difficult, some care must be taken with respect to punctuation and the conjunction rules. Next, we reorient some of the edges according to information in the lexical categories. A language for specifying these

instructions using variables and indices is given by Clark et al. (2002). This process is shown in figures 4.1, 4.2 and 4.3[10].

We used the CCG derivation-to-dependency converter `generate`, included in the C&C tools package, which is included with the Clark and Curran parser, to convert the output of the Petrov parser to dependencies. Other than a CCG derivation, their system requires only the lexicon of edge reorientation instructions, and methods for converting the unrestricted rules of CCGbank into the function argument relations. An important note for the purpose of comparison is that this system does not depend on their parser.

An unlabelled dependency is correct if the ordered pair of words is correct. A labelled dependency is correct if the ordered pair of words is correct, the head word has the correct category, and the position of the category that is the head of that dependency is correct. Table 4.13 shows accuracies from the Petrov parser trained on CCGbank, along with accuracies for the Clark and Curran parser. We only show accuracies for the Petrov parser trained on the original version of CCGbank because the dependency converter cannot currently generate dependencies for featureless derivations.

The relatively poor coverage of the Petrov parser is due to the failure of the dependency converter to output dependencies from valid CCG derivations. However, the coverage of the dependency converter is actually lower when run on the gold standard derivations, indicating that this coverage problem is not indicative of inaccuracies in the Petrov parser. Due to the difference in coverage, we again evaluate the top two parsers on only those sentences where they both generate dependencies, and report those results in tables 4.14 and 4.15. The Petrov parser has better results by a statistically significant margin for both labelled and unlabelled recall, and unlabelled F-score.

[10]We have reversed the directions of dependencies from Clark et al. (2002) to be in line with the dependency parsing literature.

4.2.5 Time and Space Evaluation

As a final evaluation, we compare the resources that are required to both train and parse with the Petrov parser on the Penn Treebank, the Petrov parser on the original version of CCGbank, the Petrov parser on CCGbank without features, and the Clark and Curran parser using both the normal-form model and the hybrid model. All training and parsing was done on a 64-bit machine with 8 dual core 2.8 GHz Opteron 8220 CPUs and 64GB of RAM. Our training times are much larger than those reported in Clark and Curran (2007b) because we report the cumulative time spent on all CPUs rather than the maximum time spent on a single CPU. Table 4.16 shows the results.

As can be seen, the Clark and Curran parser has similar training times to, although significantly greater RAM requirements than, the Petrov parsers. In contrast, the Clark and Curran parser is significantly faster than the Petrov parsers, which we hypothesize may be attributed to the degree to which Clark and Curran have optimized their code, their use of `C++` as opposed to `Java`, and their use of a supertagger to prune the lexicon.

Chapter 5

A Lambek Categorial Grammar Corpus

In this chapter, we will argue for the superiority of Lambek Categorial Grammar (LCG) over Combinatory Categorial Grammar (CCG) as the grammar formalism on which to base a practical parser. Our general argument will be that LCG provides greater transparency between the three representations of syntax, semantics, and dependency structures than CCG, or in other words, that converting between these three representations within LCG is simpler and more straightforward. More specifically, our arguments will fall into two categories. First, we will argue that the correspondence between proof trees and proof nets in LCG allows for a single LCG derivation to take the place of both the phrase-structure tree and the dependency structure that is required by CCG. Second, we will analyze a number of linguistic constructions, which require non-categorial rules in CCG. We will show the problems that such rules cause in building a semantic term from the syntax, and provide LCG derivations that allow for greater transparency between the syntax and semantics within LCG. The latter could also be applied to the CCG of CCGbank and the Clark and Curran parser, but the former is unique to LCG.

In section 5.1, we will give our analysis of the structures found in CCGbank, and show

that proof nets offer a superior representation of syntax over the dual representation of phrase-structure trees and dependency structures. In section 5.2, we will discuss associativity in categorial grammar, and argue that the full associativity of LCG is superior to the partial associativity of CCG. Section 5.3 will analyze the linguistic constructions of relative clauses, coordination and sentential gapping. We will also demonstrate the problems that non-categorial rules cause for generating semantic terms, and provide LCG derivations that solve those problems. Next, section 5.4 details our method for semi-automatically applying our new derivations to the whole of CCGbank to generate a corpus of LCG derivations. Then, section 5.5 evaluates the new corpus, LCGbank, using three methods from the literature. Finally, section 5.6 describes the application of the algorithm from chapter 3 to our newly-built corpus and analyzes its running time in practice.

5.1 Proof Nets as Dependency Structures

CCGbank is a corpus consisting of most of the sentences found in the Wall Street Journal section of the Penn Treebank (Marcus et al., 1994), annotated with CCG derivations and dependency structures. The CCG derivations were semi-automatically translated from the original Penn Treebank annotations, and the dependency structures were derived from the CCG derivations.

In this section, we will give a sentence from CCGbank whose CCG phrase-structure derivation consists of only rules that abide by the principles of categorial grammars. Our interest in such a sentence lies in the fact that the complete lack of non-categorial rules means that it has an essentially identical derivation in LCG. We will show that the proof net representation of its LCG derivation and the dependency structure from CCGbank are very closely related, demonstrating that an LCG derivation provides the same information as both the phrase-structure tree and the dependency structure in

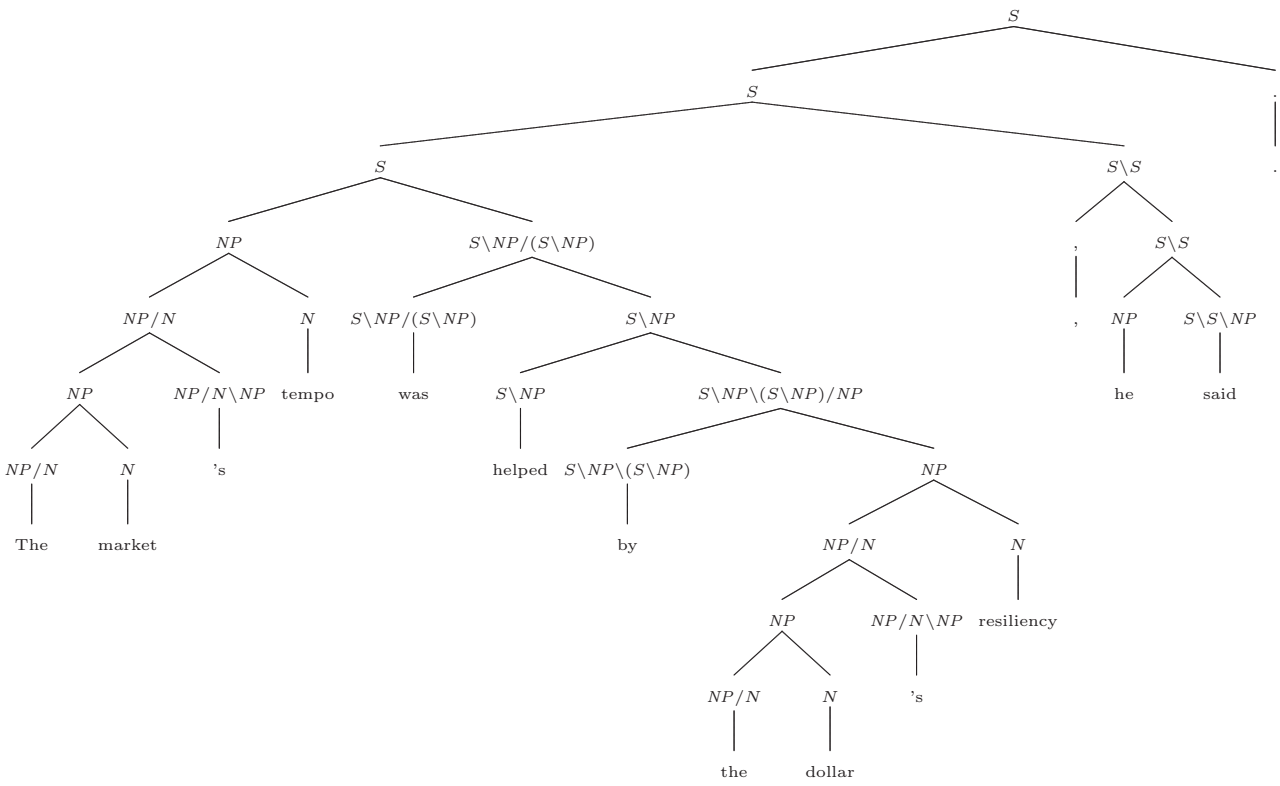


Figure 5.1: The CCGbank phrase-structure tree for sentence 0351.4.

CCGbank.

Figure 5.1 shows the CCGbank phrase-structure tree for sentence 0351.4 from CCGbank. CCGbank attaches features to some atoms, which we have omitted here, because we demonstrated in section 4.2.3 that such features are superfluous. Sentence 0351.4 is a typical sentence found in CCGbank, except for the fact that it contains no non-categorial rules. For this reason, and also because it contains no crossing rules, its CCG derivation can be modified by removing the rules for punctuation[1] to become an LCG derivation, as shown in its proof tree representation in figure 5.2. The corresponding proof net

[1]For the purposes of our analysis in this chapter, we will ignore punctuation in LCG derivations. However, punctuation that does not participate in the parse can be easily handled by a chart parser based on proof nets, such as the algorithm of chapter 3. This even includes ambiguity between non-participating punctuation and punctuation with grammatical function.

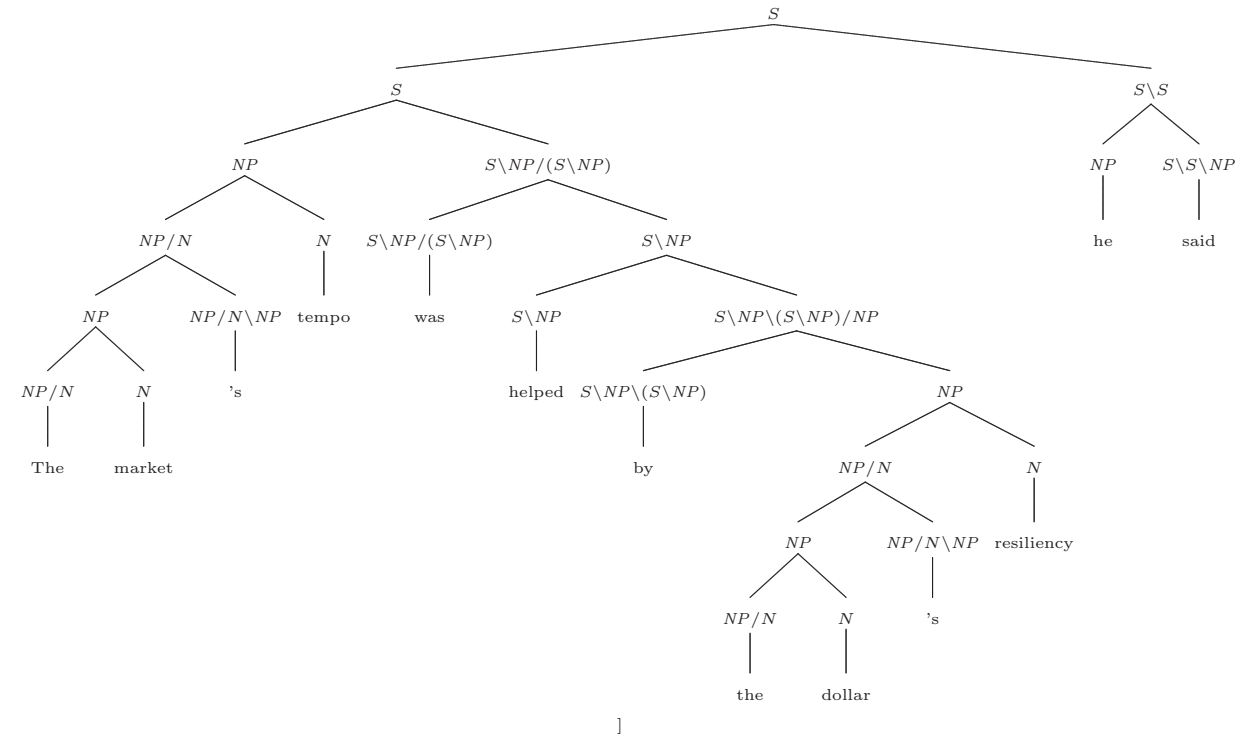


Figure 5.2: The LCG proof tree equivalent of the CCGbank phrase-structure tree for sentence 0351.4.

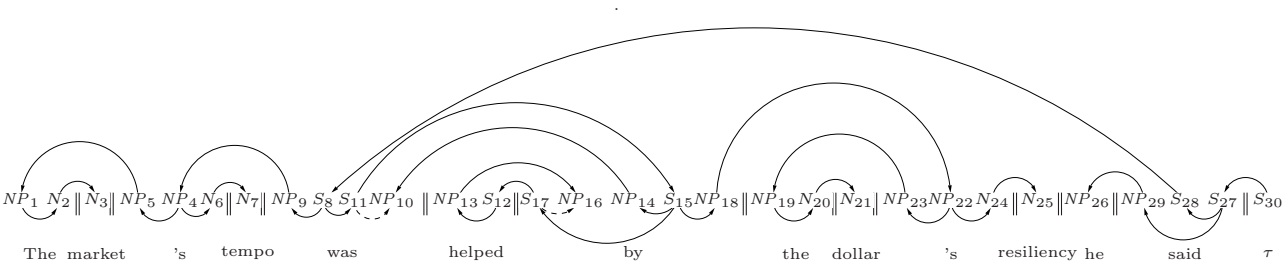


Figure 5.3: The LCG proof net corresponding to the proof tree in figure 5.2.

representation of that proof tree is shown in figure 5.3.

Alongside the phrase-structure trees found in CCGbank, there are also dependency structures included for each sentence. These dependency structures are important to CCG parsing, to the extent that they are the primary metric upon which the Clark and Curran parser is evaluated (Clark and Curran, 2007a) and the basis of its probability model (Clark and Curran, 2007b). Figure 5.4 shows the dependency structure from

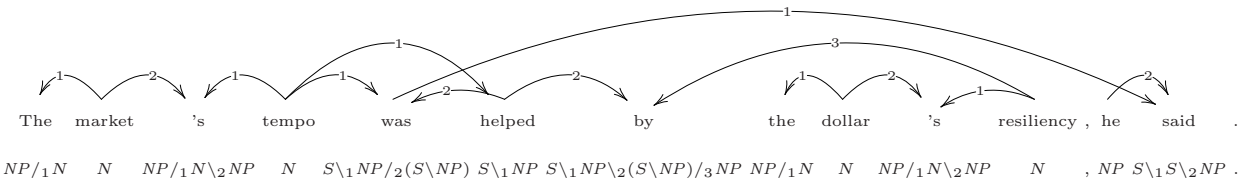


Figure 5.4: The CCGbank dependency structure for sentence 0351.4.

CCGbank for sentence 0351.4, using the same notation as in section 2.4.1.

The important point concerning the two structures shown in figures 5.3 and 5.4 is that they capture essentially the same structure. Each arc, or dependency, in the dependency structure corresponds to an arc, or link, in the proof net for the sentence.

For example, the dependency arc specifying that the word "market" is the first argument of the word "The" is derived from the fact that, in the phrase-structure tree in figure 5.1, there is an application rule combining the categories for those two words. This is precisely the relationship captured by the link from N_2 to N_3 in the proof net representation of the LCG derivation in figure 5.3.

Of the dependencies in the dependency structure for this sentence, all but three of them have a perfect representation in the LCG proof net for the sentence. The three dependencies that do not, namely the first argument of "helped", the second argument of "was" and the third argument of "by", are represented by paths in the proof net rather than single links. If desired, these dependencies can be easily generated from a proof net by building a post-processor that performs lexical analysis to convert paths into edges.

Figure 5.5 gives the complete correspondence for this sentence. Dependencies appear on the left, and are labelled by the head and the number of its argument. Proof net links appear on the right, and are labelled by their source and target in the proof net.

This correspondence demonstrates that the relationship between proof trees and proof nets in LCG is similar to the relationship between CCG phrase-structure trees and the dependencies of CCGbank. However, CCG is fundamentally a phrase-structure grammar

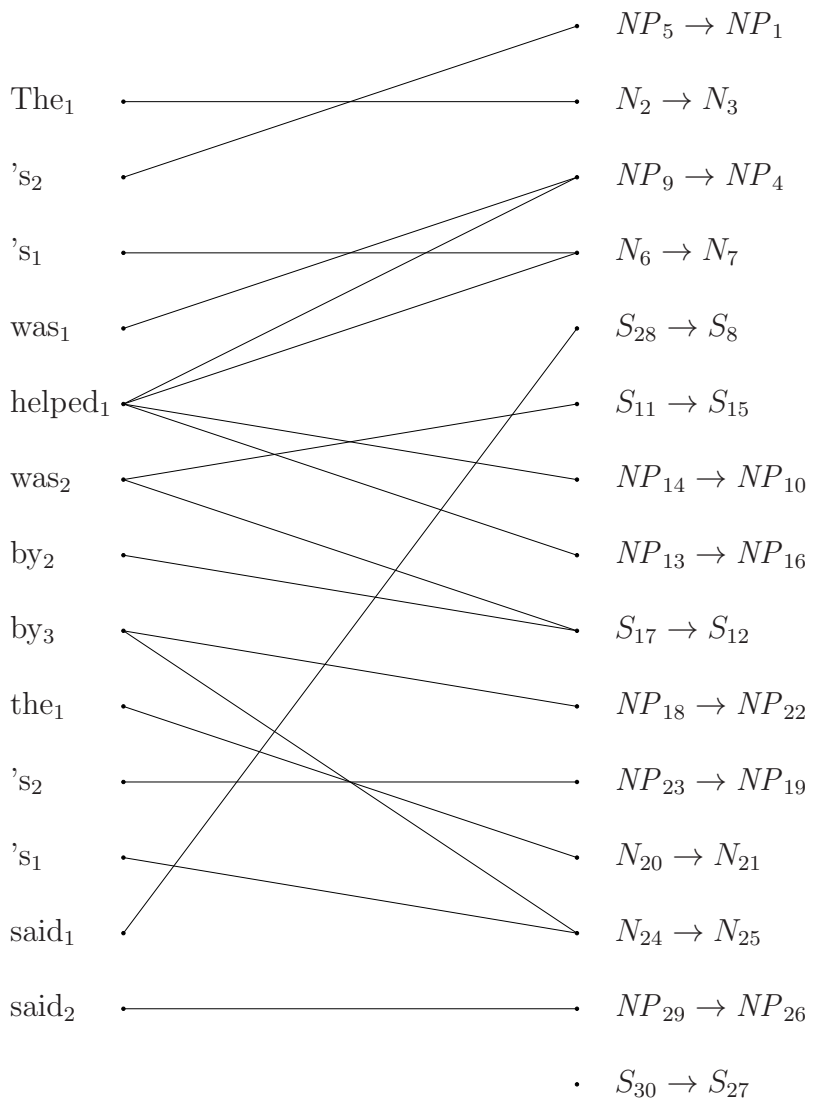


Figure 5.5: The correspondence between the dependencies in the dependency structure and the links in the proof structure for sentence 0351.4.

formalism, due to its use of a list of combinators. This means that a parser, such as the Clark and Curran parser, that is interested in using dependencies as a representation must maintain both phrase-structure trees and dependency structures internally and convert between the two. In contrast, LCG is a formalism that most naturally uses proof nets as its representation. Therefore, a parser based on LCG can maintain only a single representation, rather than two.

5.2 Associativity in Categorial Grammar

A primary motivating factor for categorial grammar is the fact that it has a flexible notion of constituency. This is important for a number of linguistic phenomena, including extraction phenomena such as wh-extraction and relative clauses and also for constructions such as so-called non-constituent coordination, as in the sentence "John loves but Bill hates Mary". The flexibility of a given categorial grammar's notion of constituency is directly related to its degree of associativity. The associativity of a function or operator refers to the importance of the order that it is applied to its arguments.

We can say that CCG is partially associative because it allows type-raising to a finitely restricted degree. CCG's type-raising allows for it to elegantly analyze linguistic constructions, such as extraction and non-constituent coordination. However, order of application rules is still important when type-raising is not possible. Because of this, CCG is partially associative, as opposed to the full associativity of LCG. This is reflected by the fact that we can use proof nets to represent LCG derivations, but not CCG derivations.

We have two pieces of evidence that the ideal degree of associativity is full associativity. First, we look to how categorial grammar parses are evaluated in practice. As we discussed in section 4.2.4, Clark and Curran evaluate their parser on the dependencies that it produces, rather than on CCG derivations. The reason behind this is that the constituent structure present in a CCG is irrelevant, and only the information expressed

by the dependencies is important. However, dependencies are fully associative in the sense that they do not specify any order of application among the dependencies.

Our second piece of evidence comes from how CCG derivations are constructed. Because of the partial associativity of CCG, it is often the case that there are a number of possible CCG derivations for a sentence that differ by the number of compositions and applications, but which all have identical dependencies and semantic terms. The Clark and Curran parser, using the normal form model, uses a normal form to choose between these derivations, where application rules are preferred over composition rules when both are possible. Again, this indicates that the constituents present in a CCG derivation are arbitrary.

This evidence leads us to conclude that the constituent structure represented in CCG derivations is irrelevant, and therefore should be abandoned if possible, as it unnecessarily adds to parser complexity. Abandoning this constituent structure requires full associativity which, in turn, requires proof nets like those in LCG. The benefit from using proof nets is that we have one representation that gives word-to-word dependencies and dictates the syntactic structure of our sentence. Having only one syntactic representation allows for a simpler corpus and a simpler parser, without sacrificing anything due to the irrelevance of the constituent structure of CCG. Furthermore, it makes analysis of the corpus much easier because our representation does not contain arbitrary structure.

5.3 Problematic Linguistic Constructions in CCGbank

In this section we will be introducing a number of examples of sentences found in CCGbank that will show the advantages of using LCG as a categorial grammar. Our general method in this section will be to provide a representative sentence from CCGbank, outline the difficulty of generating a semantic term from its phrase-structure tree in CCGbank,

and show how LCG can help in regards to that difficulty. Our LCG derivations will be the result of *lexicalizing* much of the grammar, or in other words, solving the problems of generating semantics by building a more complex lexicon to accompany the simpler rule system of LCG.

5.3.1 Relative clauses

There are a large number of relative clause constructions found in the sentences of CCGbank. We will use relative clauses to demonstrate the increased transparency between syntax and dependency structures in LCG, and also the advantage of associativity in LCG. A relative clause is a phrase that modifies a noun phrase, such as the following:

(5.1) an event that he said would temporarily dilute earnings[2]

The CCG phrase-structure tree and the dependency structure from CCGbank of (5.1) can be seen in figures 5.6 and 5.7. There are two primary difficulties with the derivation in figure 5.6, both relating to the partial associativity of CCG.

The first problem is that the phrase "said would temporarily dilute earnings" receives the category $S\backslash NP/NP$, but CCG contains no rules for combining this category directly to the category NP for the word "he". Instead, a type-raising rule must be applied to the category NP before a composition rule can combine the result with the category $S\backslash NP/NP$. This is problematic because type-raising is an extra rule that requires additional parsing resources in terms of space and time.

The second, more subtle, problem is that the phrase-structure tree shown in figure 5.6 for the phrase "he said would temporarily dilute earnings" is one of several possible phrase-structure trees for this phrase. Figure 5.6 shows a right-branching phrase-structure tree, whereas an alternative phrase-structure tree exists that is left-branching,

[2] A fragment of CCGbank sentence 0317.33.

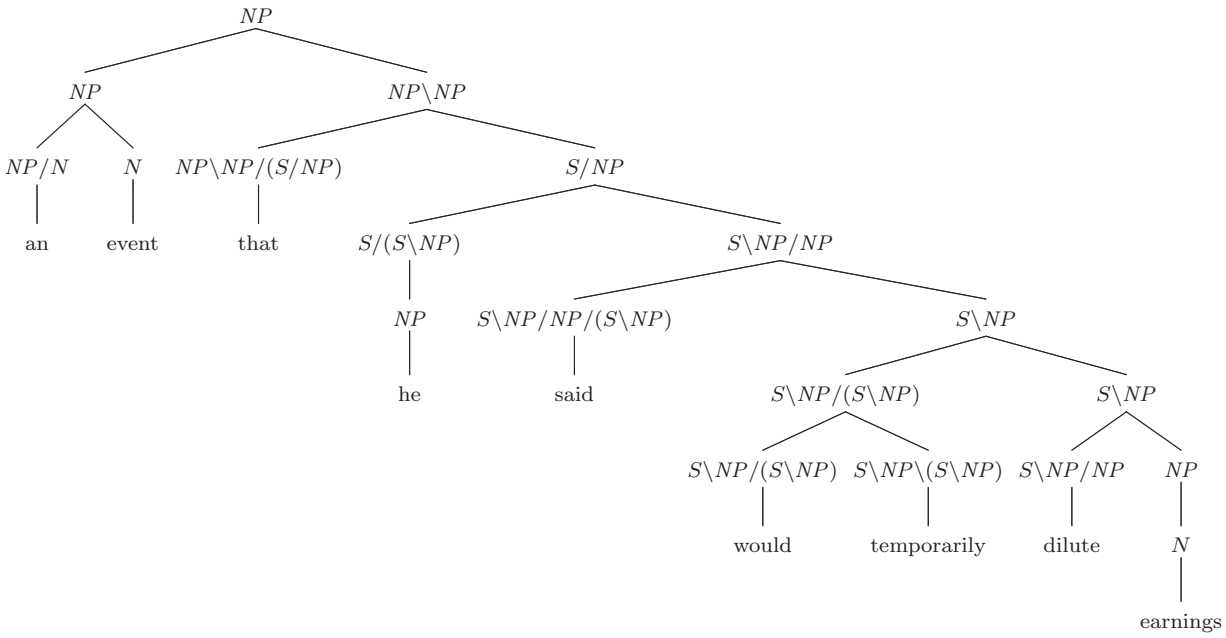


Figure 5.6: The CCGbank phrase-structure tree for the relative clause of (5.1).

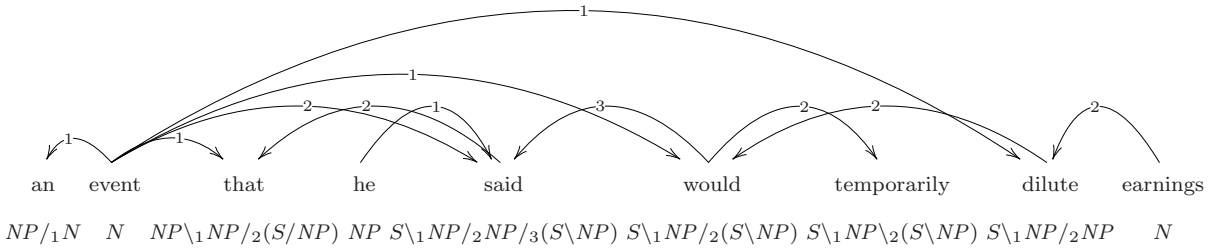


Figure 5.7: The CCGbank dependency structure of the relative clause of (5.1).

as shown in figure 5.8. This problem, referred to as *spurious ambiguity*, is the existence of two distinct phrase-structure trees that are not indicative of two distinct semantic interpretations, since both the dependencies and the semantic term for the two phrase-structure trees are identical. Spurious ambiguity is a fundamental problem with phrase-structure representations of categorial grammar, and has been dealt with in the CCG literature by enforcing normal forms for the phrase-structure trees (Eisner, 1996, Hockenmaier and Steedman, 2007).

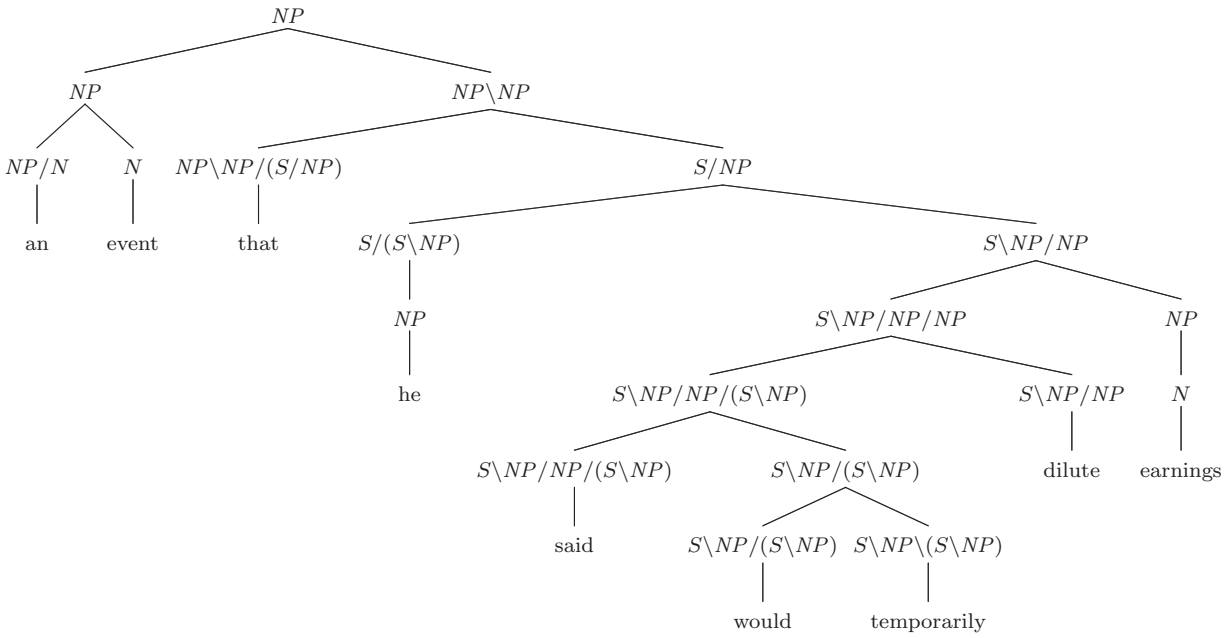


Figure 5.8: An alternative CCGbank phrase-structure tree for the relative clause of (5.1).

Both of these problems stem from the fact that CCG without type-raising is fundamentally non-associative. The solution within the CCG framework is to add type-raising and require a normal form for CCG derivations. In particular, CCGbank always chooses the right-branching structure in the presence of spurious ambiguity. However, this additional structure is unnecessary and adds additional complexity to the parser without any apparent benefit.

The LCG derivation is shown in figure 5.9 with the lexicon in table 5.1 with the resulting semantic term of $\exists y.event(y) \wedge say(he, would(temporarily(dilute(y, earnings))))$. The LCG derivation does not require explicit type-raising rules and does not have any spurious ambiguity. The categorial semantics for the LCG derivation in figure 5.9 is identical to that of the CCG derivation in figure 5.6, while the dependencies obtained from the CCG derivation can be obtained from the LCG derivation by deleting some of the edges in the proof net and reorienting some edges, as was done when deriving the dependencies from the CCG phrase-structure tree. The LCG derivation is superior

Word	Syntactic Category	Semantic Term
an	NP/N	$\lambda x.\exists y.x(y)$
event	N	$\lambda x.event(x)$
that	$NP\backslash NP/(S/NP)$	$\lambda xy.y \wedge x(y)$
he	NP	he
said	$S\backslash NP/NP/(S\backslash NP)$	$\lambda xyz.say(z, x(y))$
would	$S\backslash NP/(S\backslash NP)$	$\lambda x.would(x)$
temporarily	$S\backslash NP/(S\backslash NP)\backslash(S\backslash NP/(S\backslash NP))$	$\lambda x.temporarily(x)$
dilute	$S\backslash NP/NP$	$\lambda xy.dilute(y, x)$
earnings	NP	$earnings$

Table 5.1: The LCG lexicon for the relative clause of (5.1).

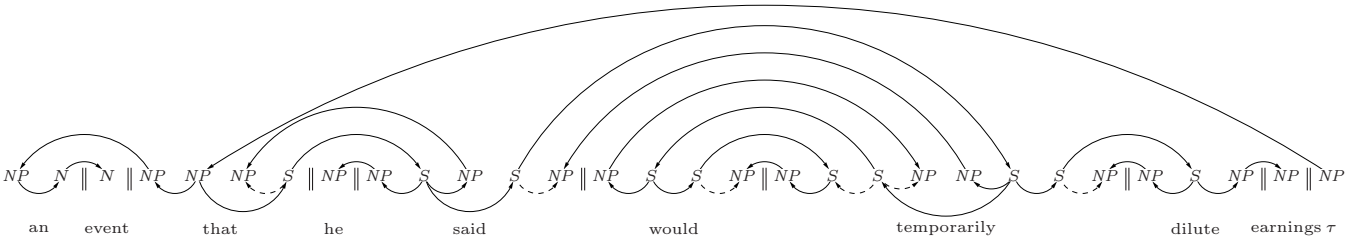


Figure 5.9: The LCG proof net for the relative clause of (5.1).

due to the fact that the syntactic representation does not contain spurious ambiguity, resulting in a closer relationship between syntax and semantics for the LCG derivation, and therefore greater transparency between its syntax and semantics.

5.3.2 Reduced relative clauses

In this section, we will introduce *reduced* relative clauses, whose derivations in CCGbank use non-categorial rules. The primary problem with such derivations is the difficulty in obtaining a categorial semantics from the syntactic representation found in CCGbank.

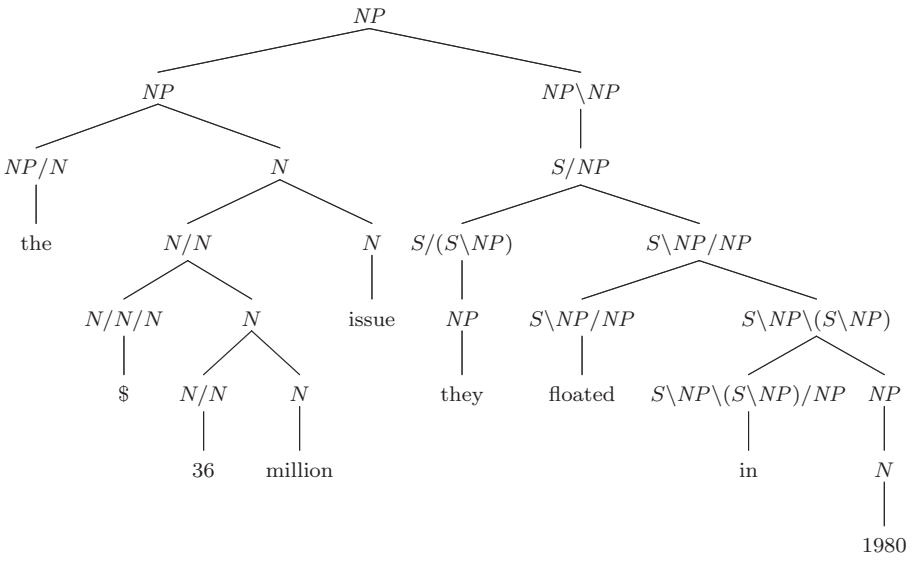


Figure 5.10: The CCGbank phrase-structure tree for the reduced relative clause of (5.2).

A reduced relative clause is similar to a relative clause except that the complementizer (i.e. the word "that") is missing:

(5.2) the $ 36 million issue they floated in 1980[3]

Reduced relative clauses are quite common and appear throughout CCGbank.

With respect to the CCGbank analysis of the relative clause from the previous section, the complementizer converted the S/NP category of the phrase "he said would temporarily dilute earnings" into the required noun modifying the $NP\backslash NP$ category. Without the complementizer category, an alternative analysis is necessary. CCGbank provides the CCG derivation in figure 5.10 for the fragment shown in (5.2).

The reason that CCGbank uses the syntactic derivation shown in figure 5.10 is that it can use essentially the same syntactic derivation, and consequently semantic term, for the phrase "they floated in 1980", regardless of whether it occurs in a reduced relative clause or not. However, using the syntactic derivation in figure 5.10 is problematic

[3]A fragment of CCGbank sentence 0339.13.

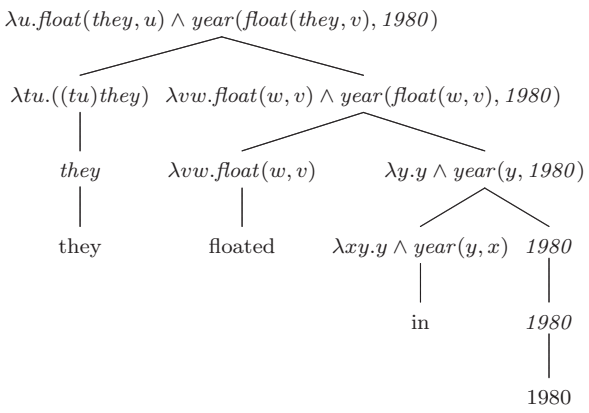


Figure 5.11: The categorial semantics for the phrase "they floated in 1980".

because generating semantics for the non-categorial rule $NP\backslash NP \rightarrow S/NP$ is difficult. Unfortunately, CCGbank does not provide semantic terms to accompany its syntactic derivations. Instead, Bos et al. (2004) details a method for generating semantics from the derivations output from the Clark and Curran parser. Figure 5.11 shows the semantics of Bos et al. (2004) for the phrase "they floated in 1980".

The problem with the resultant semantic term in figure 5.11 is that it is the semantics for a sentence missing a noun phrase (i.e. a phrase with syntactic category S/NP). We must obtain, from that semantic term, the semantics for a *noun phrase* missing a noun phrase (i.e. a phrase with syntactic category $NP\backslash NP$). Figure 5.12 shows an example of a semantic derivation for the syntactic derivation in figure 5.10.

The primary problem with the semantic derivation shown in figure 5.12 is how to obtain the term $\lambda s.s \wedge \mathit{float}(\mathit{they}, r) \wedge \mathit{year}(\mathit{float}(\mathit{they}, r), \mathit{1980})$ from the term $\lambda u.\mathit{float}(\mathit{they}, u) \wedge \mathit{year}(\mathit{float}(\mathit{they}, u), \mathit{1980})$, or, in other words, how to generate the semantics for the unary rule $NP\backslash NP \rightarrow S/NP$. The theoretical categorial grammar literature offers nothing to help with this conversion, due to the fact that the rule $NP\backslash NP \rightarrow S/NP$ is not a rule of any categorial grammar from that literature. Of course, we can specify a system for performing the necessary conversion, by deleting certain segments of the semantic term

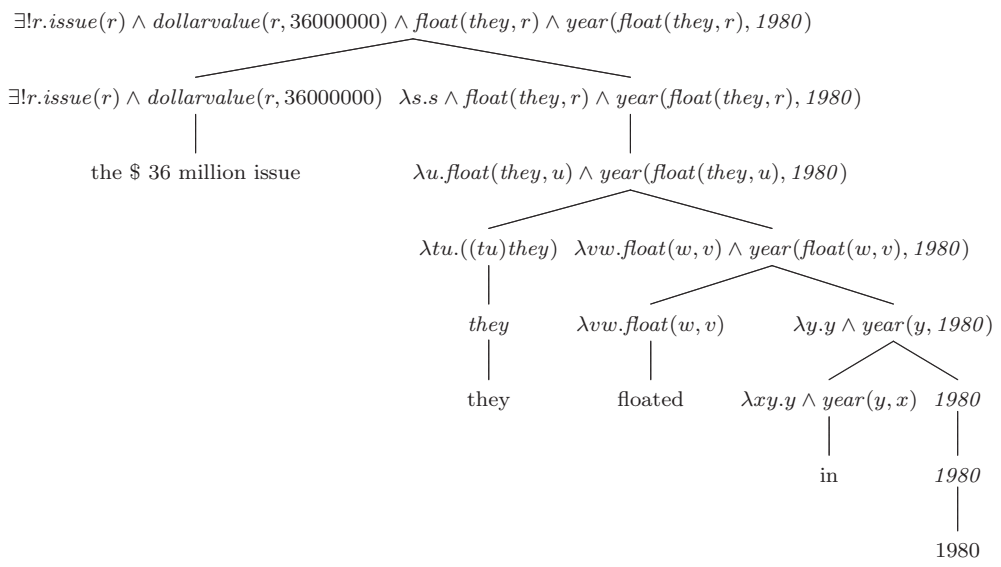


Figure 5.12: The semantics for the CCGbank derivation in figure 5.10.

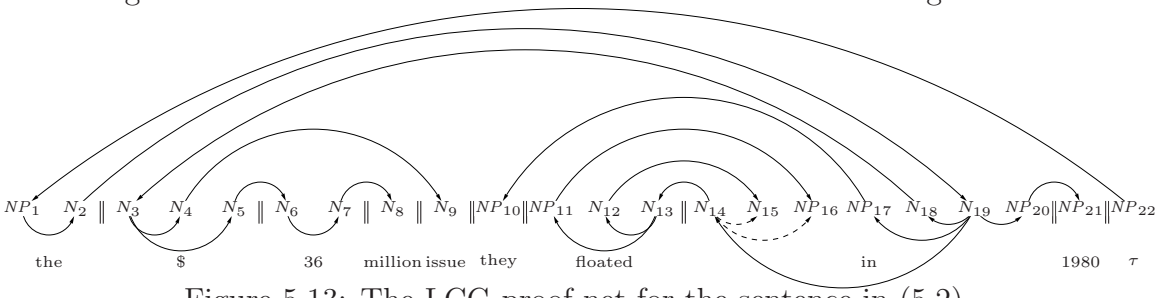


Figure 5.13: The LCG proof net for the sentence in (5.2).

and traversing the tree to find necessary variable substitutions. However, such a system for generating semantics will not generalize to more complex sentences and is bound to produce incorrect semantic terms on unseen data.

Within LCG, we cannot use such non-categorial rules and we will argue for the syntactic derivation shown in figure 5.13, using the lexicon in table 5.2.

In the lexicon in table 5.2, the categories for the words are identical to those found in figure 5.10, except for the categories for "floated" and "in". The category for the word "floated" has been altered so that, in the LCG derivation, it takes an NP and produces the noun modifier category $N\backslash N$. Also, the category for the word "in" has been adjusted, so that it can modify the new category for "floated".

Word	Syntactic Category	Semantic Term
the	NP/N	$\lambda x.\exists!y.x(y)$
$	$N/N/N$	$\lambda xyz.y(z) \wedge dollarvalue(z,x)$
36	N/N	$\lambda x.36 * x$
million	N	1000000
issue	N	$\lambda x.issue(x)$
they	NP	*they*
floated	$N\backslash N\backslash NP$	$\lambda xyz.float(x,z) \wedge y(z)$
in	$N\backslash N\backslash NP\backslash(N\backslash N\backslash NP)/NP$	$\lambda xy.y \wedge year(y,x)$
1980	NP	*1980*

Table 5.2: The LCG lexicon for the sentence in (5.2).

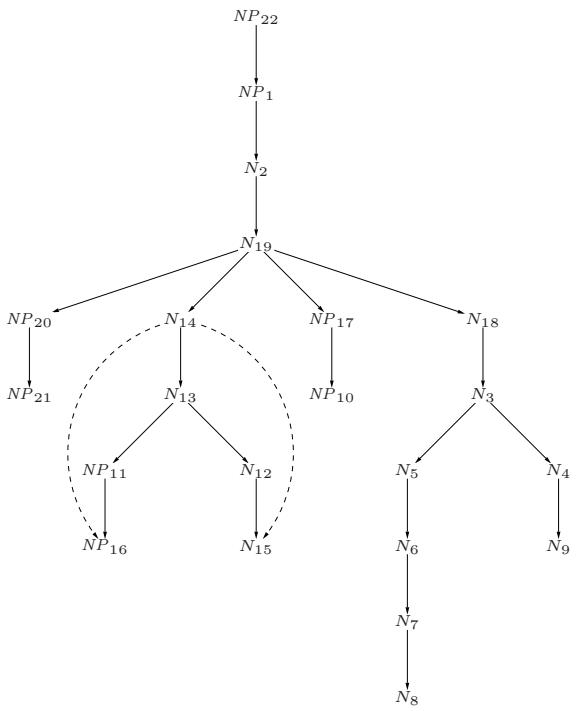


Figure 5.14: A tree view of the LCG proof net in figure 5.13.

The semantics for this LCG derivation can be obtained from the semantic terms for the words and the proof net. More background on deriving the semantic term from a proof net can be found in (Roorda, 1991). To calculate the semantic term, it can be useful to view proof nets using a tree view, as shown in figure 5.14. The matching edges, or rather those above the atoms in the proof net, specify applications that occur between lambda terms. The dotted Lambek edges specify functional abstractions that occur in the derivation of the semantic term.

We will use the variable a to refer to the vertex NP_{16} and the variable b to refer to the vertex N_{15}. We will derive the semantics by first determining the applications that are specified by the proof net, using dummy variables to correspond to the semantics for each word. We will use the variable c to represent the semantics for the word "the", the variable d to represent the semantics for the word "$" and so on until k is used to represent the word "1980". The tree view is useful for calculating the semantic term because the semantic term for a word or phrase depends only on those vertices lower down in the tree view than the atoms for the words in the phrase.

The semantics for the phrase "36 million" is obtained by observing that the relevant atoms are those below and including N_6 in figure 5.14. The only matching edge in that subgraph is the edge from N_7 to N_8, which yields the semantics (ef) for the phrase "36 million", or in other words, the semantic term for 36 applied to the semantic term for "million". Similarly, the phrase "$ 36 million issue" yields the semantic term $((d(ef))g)$.

The semantic term for "floated" is determined by those vertices lower than or equal to N_{13} in figure 5.14. There are two matching edges indicating two applications. The resultant semantic term for the word "floated" is $((ia)b)$.

The semantic term for the phrase "$ 36 million issue they floated in 1980" is more complex. Because the syntactic category for "in" contains two Lambek edges, we must properly determine where the lambda abstractions would appear in the semantic term for the word "in". Otherwise, we perform the applications in the order of the arguments.

The resultant semantic term for the phrase "$ 36 million issue they floated in 1980" is $((\lambda ab.((jk)((ia)b))h)((d(ef))g))$, obtained by copying the semantic terms for "floated" and "$ 36 million issue" in the appropriate locations. This can be β-reduced to the term $((jk)((ih)((d(ef))g)))$.

We can determine the semantic term for the entire sentence by applying the semantic term for "the" to the term from above. The result is $(c((jk)((ih)((d(ef))g))))$. Substituting the actual semantic terms in for the dummy variables and β-reducing yields the derivation in figure 5.15.

The advantage of this LCG derivation is that we achieve the correct semantic term without needing to resort to ad hoc semantic term rewriting. This leads to a greater transparency between the syntax and the semantics for LCG, since the semantic term for the sentence follows directly from the semantic terms in the lexicon and from the syntactic derivation in the form of proof nets.

The disadvantage of our approach is the increased size of the lexicon, which will require more sophisticated mechanisms for assigning supertags during the parsing process. However, this disadvantage is offset by the increased usefulness of the lexicon. The lexicon for LCG contains more of the contents of the grammar, which allows grammar analysis to focus exclusively on the lexicon rather than dividing its attention between the lexicon and the rule system. Furthermore, in section 5.4.3 we train the Petrov parser on our corpus of LCG derivations, to annotate the left-out sentences, and the greater complexity in the lexicon is easily learned.

5.3.3 Coordination

Coordination structures are linguistic structures involving a coordinator such as "and", "or" or "but", together with the two conjuncts that it conjoins. CCGbank contains a very large number of such constructions, of which a typical example is the following:

$$(c((jk)((ih)((d(ef))g))))$$

$$\rightarrow(c((jk)((ih)((d((\lambda x.36 * x)1000000))g))))$$

$$\rightarrow(c((jk)((ih)((d(36 * 1000000))g))))$$

$$\rightarrow(c((jk)((ih)((d36000000)g))))$$

$$\rightarrow(c((jk)((ih)(((\lambda xyz.y(z) \wedge dollarvalue(z, x))36000000)g))))$$

$$\rightarrow(c((jk)((ih)(\lambda yz.y(z) \wedge dollarvalue(z, 36000000)g))))$$

$$\rightarrow(c((jk)((ih)(\lambda yz.y(z) \wedge dollarvalue(z, 36000000)(\lambda x.issue(x))))))$$

$$\rightarrow(c((jk)((ih)(\lambda z.issue(z) \wedge dollarvalue(z, 36000000)))))$$

$$\rightarrow(c((jk)((ih)(\lambda z.issue(z) \wedge dollarvalue(z, 36000000)))))$$

$$\rightarrow(c((jk)(((\lambda xyz.float(x, z) \wedge y(z))they)(\lambda z.issue(z) \wedge dollarvalue(z, 36000000)))))$$

$$\rightarrow(c((jk)((\lambda yz.float(they, z) \wedge y(z))(\lambda z.issue(z) \wedge dollarvalue(z, 36000000)))))$$

$$\rightarrow(c((jk)(\lambda z.float(they, z) \wedge issue(z) \wedge dollarvalue(z, 36000000))))$$

$$\rightarrow(c(((\lambda xy.y \wedge year(y, x))1980)(\lambda z.float(they, z) \wedge issue(z) \wedge dollarvalue(z, 36000000))))$$

$$\rightarrow(c((\lambda y.y \wedge year(y, 1980))(\lambda z.float(they, z) \wedge issue(z) \wedge dollarvalue(z, 36000000))))$$

$$\rightarrow(c(\lambda z.float(they, z) \wedge issue(z) \wedge dollarvalue(z, 36000000) \wedge year(float(they, z), 1980)))$$

$$\rightarrow(\lambda x.\exists!y.x(y)(\lambda z.float(they, z) \wedge issue(z) \wedge dollarvalue(z, 36000000) \wedge year(float(they, z), 1980)))$$

$$\rightarrow\exists!y.((\lambda z.float(they, z) \wedge issue(z) \wedge dollarvalue(z, 36000000) \wedge year(float(they, z), 1980))y)$$

$$\rightarrow\exists!y.float(they, y) \wedge issue(y) \wedge dollarvalue(y, 36000000) \wedge year(float(they, y), 1980)$$

Figure 5.15: The LCG semantic derivation for the sentence in (5.2).

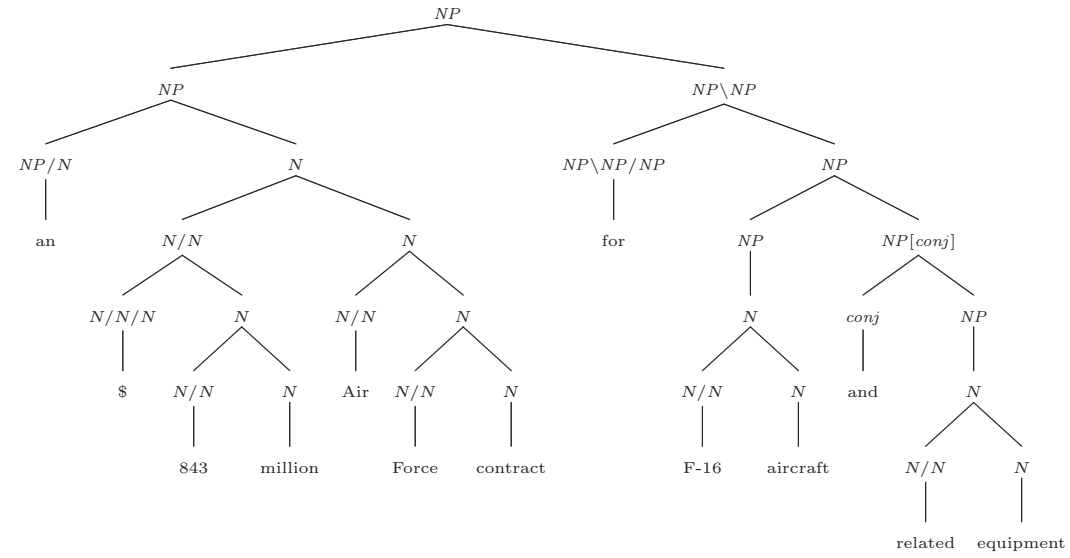


Figure 5.16: The CCGbank phrase-structure tree for (5.3).

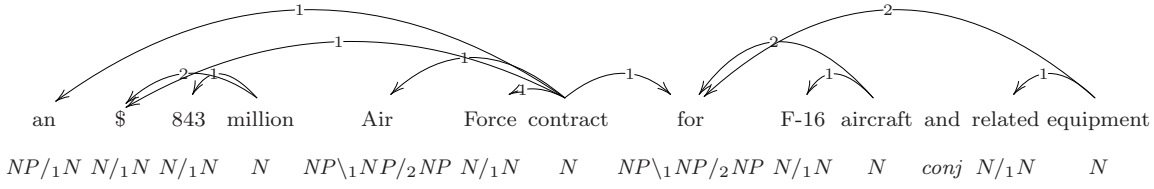


Figure 5.17: The CCGbank dependency structure for (5.3).

(5.3) an $ 843 million Air Force contract for F-16 aircraft and related equipment[4]

The coordinator in (5.3) is the word "and", and the two conjuncts are the noun phrases "F-16 aircraft" and "related equipment". The CCGbank phrase-structure tree for (5.3) is shown in figure 5.16 and the CCGbank dependency structure is shown in figure 5.17.

As is shown in figure 5.16, CCGbank handles coordination by assigning the category *conj* to the coordinator "and", and then forming a constituent with the right conjunct, and then finally a constituent together with both conjuncts. This is in accordance with the treatment of coordination in the CCG literature in general (Steedman, 2000). The primary problem with such an analysis is that it creates opacity between the syntax and

[4]A fragment of CCGbank sentence 0330.1.

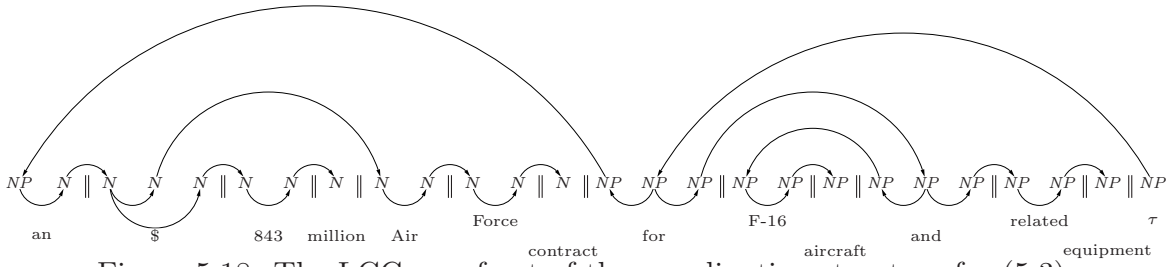


Figure 5.18: The LCG proof net of the coordination structure for (5.3).

any semantic representation, since there is little guidance for building dependency structures or semantic terms. In particular, this does not take advantage of the fact that we are working within categorial grammar, and instead resorts to an approach that has more in common with typical phrase-structure approaches. Therefore, additional mechanisms are required that can handle the *conj* category to generate the correct dependencies and semantic terms. Such a mechanism is straightforward in this simple case, but becomes increasingly convoluted in the case of *unlike* coordination.

In contrast, the LCG derivation for such a construction is shown in figure 5.18, along with the lexicon in table 5.3. At this point, we should note that the semantics in this section are not always ideal. In particular, the analysis of the phrase "Air Force" as two independent words in the syntax forces the semantics into the awkward form found in the semantic term for the sentence. This analysis is due to the syntax found in CCGbank (and also, in the Penn Treebank). As our goal in this section is not to fix all of the problems in CCGbank, but merely those related to the transparency between syntax and semantics, we will be unconcerned with such problems.

The advantage of this approach is that the relationship between the coordinator "and" and its two conjuncts is made explicit in the syntax. As a result, there is a high degree of transparency between the syntax and the semantics. From the lexicon and the proof net, we can calculate the following semantic term for the sentence: $\exists y.for(air(force(contract(y))) \wedge dollarvalue(y, 843000000), F16(aircraft) \wedge related(equipment))$. Without the explicit syntax of LCG, the relationship between the syntax and the seman-

Word	Syntactic Category	Semantic Term
an	NP/N	$\lambda x.\exists y.x(y)$
$	$N/N/N$	$\lambda xyz.y(z) \wedge dollarvalue(z,x)$
843	N/N	$\lambda x.843 * x$
million	N	1000000
Air	N/N	$\lambda xy.air(x(y))$
Force	N/N	$\lambda xy.force(x(y))$
contract	N	$\lambda x.contract(x)$
for	$NP\backslash NP/NP$	$\lambda xy.for(x,y)$
F-16	NP/NP	$\lambda x.F16(x)$
aircraft	NP	$aircraft$
and	$NP\backslash NP/NP$	$\lambda xy.x \wedge y$
related	NP/NP	$\lambda x.related(x)$
equipment	NP	$equipment$

Table 5.3: The LCG lexicon for the coordination structure in (5.3).

tics is much less clear.

In the example above, the two conjuncts were both noun phrases, but the transparency between syntax and semantics becomes weaker as the conjuncts become more complex. We will give two additional examples with more complex syntactic categories. Each example will illustrate a problem with generating semantics from the phrase-structure trees of CCGbank.

The first example involves coordination of adjectives, or in other words, coordination where the two conjuncts each have syntactic category N/N. CCGbank contains more than 1,000 examples of adjectival coordination. For example:

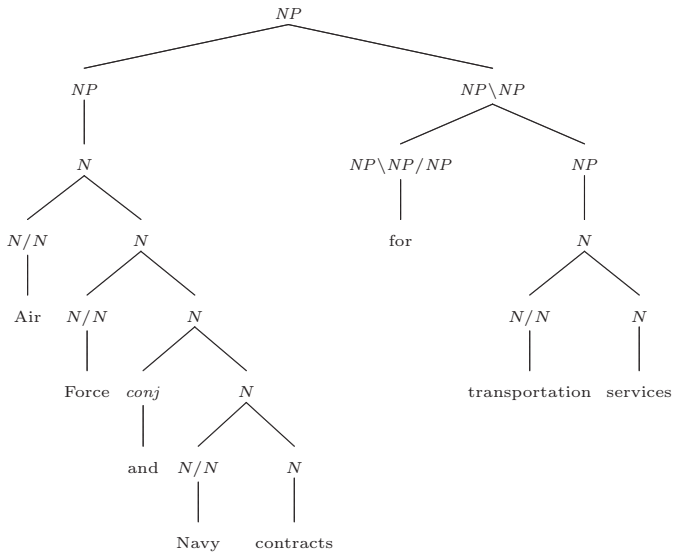


Figure 5.19: The CCGbank phrase-structure tree for (5.4).

(5.4) Southern Air transport won $ 47.5 million in Air Force and Navy contracts for transportation services [5]

The CCGbank phrase-structure tree for the adjectival coordination of (5.4) is shown in figure 5.19.

The phrase-structure tree shown in figure 5.19 has the same problems surrounding the category *conj* as the coordination structure with noun conjuncts had in figure 5.16. However, even with a special semantics defined for the word "and", generating a correct term representation from the structure in figure 5.19 will be difficult. The problem arises from building a structure in which both the semantic term for "Air Force" and the semantic term for "Navy" can be applied to the semantic term for "contracts". The phrase "Navy contracts" must be assigned some semantic term because it is a constituent in the phrase-structure tree, but then, the coordinator "and" must extract the semantics of "contracts" from that representation if it is to build a compositional semantics for the

[5] A fragment of CCGbank sentence 0330.3.

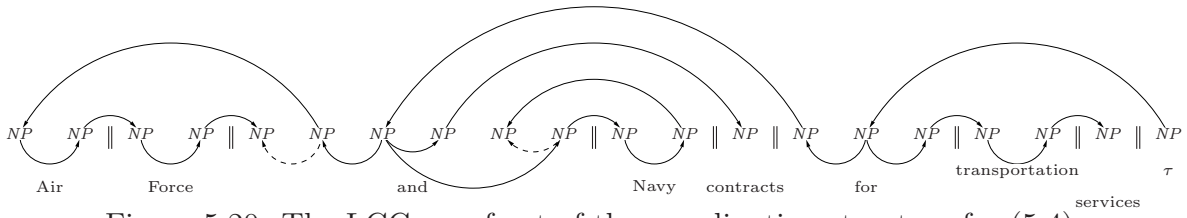


Figure 5.20: The LCG proof net of the coordination structure for (5.4).

Word	Syntactic Category	Semantic Term
Air	NP/NP	$\lambda x.air(x)$
Force	N/NP	$\lambda x.force(x)$
and	$NP/NP\backslash(NP/NP)/(NP/NP)$	$\lambda xyz.x(z) \wedge y(z)$
Navy	NP/NP	$\lambda x.navy(x)$
contracts	N/N	$contracts$
for	$NP\backslash NP/NP$	$\lambda xy.for(x, y)$
transportation	NP/NP	$\lambda x.transportation(x)$
services	NP	$services$

Table 5.4: The LCG lexicon for the coordination structure in (5.4).

whole phrase.

LCG solves this problem by specifying a different syntax for sentences involving adjectival coordination. The LCG derivation for the relevant fragment of (5.4) is shown in figure 5.20 with the lexicon in table 5.4.

The advantage of the LCG derivation over the CCG derivation is that the semantics for the word "and" explicitly specifies how to obtain a semantic term for the phrase "Air Force and Navy", which can then be applied to the semantics for the word "contracts". In particular, we can obtain the derivation of the semantic term for the phrase "Air Force and Navy contracts" shown in figure 5.21 directly from the proof net and the lexicon

$$
\begin{aligned}
&(((cd)(ab))e) \\
\rightarrow&(((cd)((\lambda x.air(x))(\lambda x.force(x))))e) \\
\rightarrow&(((cd)(\lambda x.air(force(x)))e) \\
\rightarrow&((((\lambda xyz.x(z) \wedge y(z))(\lambda x.navy(x)))(\lambda x.air(force(x))))e) \\
\rightarrow&(((\lambda yz.navy(z) \wedge y(z))(\lambda x.air(force(x))))e) \\
\rightarrow&((\lambda z.navy(z) \wedge air(force(z)))e) \\
\rightarrow&((\lambda z.navy(z) \wedge air(force(z)))contracts) \\
\rightarrow&navy(contracts) \wedge air(force(contracts))
\end{aligned}
$$

Figure 5.21: The derivation of the LCG semantic term for the phrase "Air Force and Navy contracts".

via variable substitution and β-reduction. As before, we will use dummy variables to temporarily refer to the semantic terms for words. We will use a for "Air" and so on until we use e for "contracts".

The LCG derivation shows a greater degree of transparency between syntax and semantics because obtaining the semantics from the syntax is a matter of following the term substitutions specified by the proof net and the lexicon.

The final type of coordination that we will discuss is *unlike* coordination. Unlike coordination is a construction where the two conjuncts do not share a syntactic type, such as in the following example:

(5.5) the bonds will come a little richer and in a larger amount[6]

[6]A fragment of CCGbank sentence 0351.31.

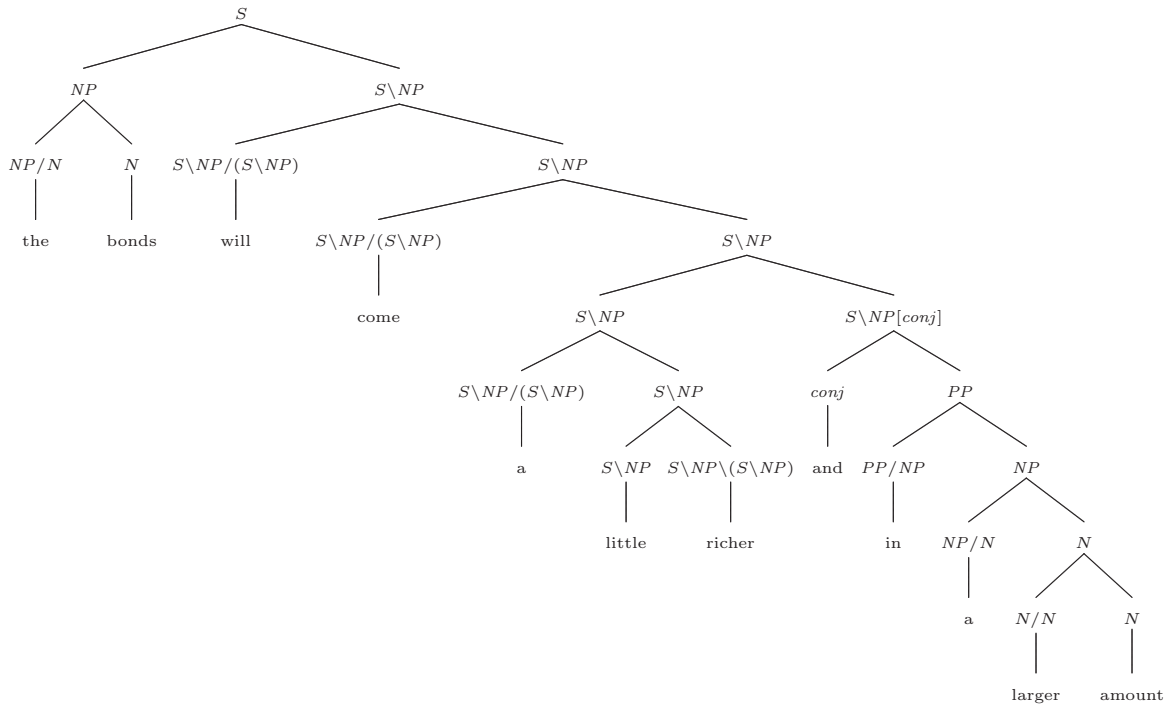


Figure 5.22: The CCGbank phrase-structure tree for (5.5).

Unlike coordination poses problems for generating semantic terms from syntax due to the fact that the coordinator cannot be assigned a semantic term that simply conjoins the terms for the conjuncts around a logical operator. The CCGbank derivation for (5.5) is shown in figure 5.22.

The difficulty with building a categorial semantics from the phrase-structure tree shown in figure 5.22 is that it is not clear from the syntax what to assign as the semantics for the word "and" to obtain the correct semantics for the whole sentence. In particular, it is not clear how to convert the semantics for the prepositional phrase "in a larger amount" into the semantics for a phrase with syntactic category $S\backslash NP$.

LCG cannot have non-categorial rules such as $S\backslash NP[conj] \rightarrow conj, PP$. Therefore, an LCG derivation must specify the precise syntactic relations between the syntactic categories. Figure 5.23 shows the syntax within LCG for (5.5) together with the lexicon

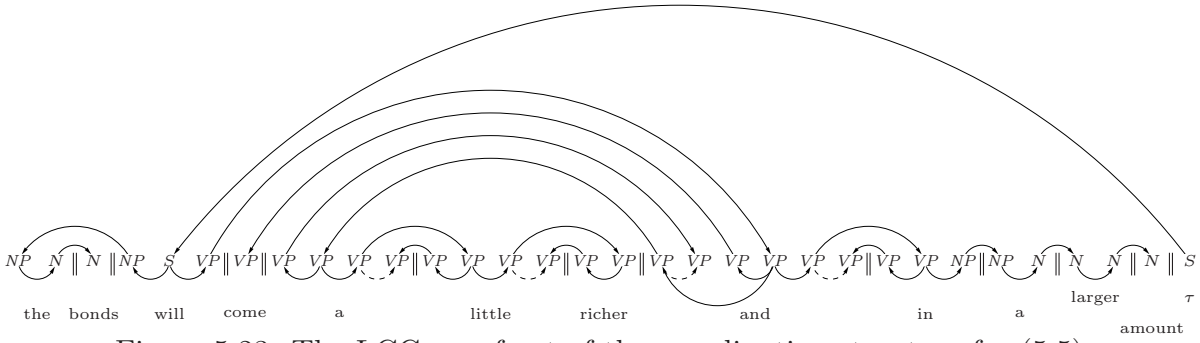


Figure 5.23: The LCG proof net of the coordination structure for (5.5).

Word	Syntactic Category	Semantic Term
the	NP/N	$\lambda x.\exists!y.x(y)$
bonds	N	$\lambda x.bonds(x)$
will	$S\backslash NP/VP$	$\lambda xy.will(x(y))$
come	VP	$\lambda x.come(x)$
a	$VP\backslash VP/(VP\backslash VP)$	$\lambda x.x$
little	$VP\backslash VP/(VP\backslash VP)$	$\lambda xy.x(y)$
richer	$VP\backslash VP$	$\lambda x.richer(x)$
and	$VP\backslash VP\backslash(VP\backslash VP)/(VP\backslash VP)$	$\lambda xyz.x(z) \wedge y(z)$
in	$VP\backslash VP/NP$	$\lambda xyz.y(z) \wedge in(z,x)$
a	NP/N	$\lambda x.\exists y.x(y)$
larger	N/N	$\lambda xy.larger(x(y))$
amount	N	$\lambda x.amount(x)$

Table 5.5: The LCG lexicon for the coordination structure of (5.5).

in table 5.5[7].

The LCG proof net in figure 5.23 specifies that the two conjuncts are verb phrase modifiers. Then, the semantic term for "and" in the lexicon specifies that the seman-

[7]We will use VP as a shorthand for $S\backslash NP$ whenever possible to ease comprehension.

tics for the two conjuncts are combined with the logical connective $\wedge$. The important difference between the LCG proof net and the CCG phrase-structure tree is that the syntax and the semantics are very closely related in the LCG derivation. This allows for a principled derivation of a correct semantic representation.

5.3.4 Sentential Gapping

A sentential gapping construction is a coordination construction where the left conjunct consists of a sentence, but the right conjunct consists of a sentence where the verb structure is missing. The following is an example from the Penn Treebank:

(5.6) Beacon Hill is for move-up town houses, and Nob Hill, for single-family homes[8]

For our purposes, we will consider the semantics of (5.6) to be identical to the sentence "Beacon Hill is for move-up town houses, and Nob Hill is for single-family homes". In other words, the word "is" is missing from the right conjunct of (5.6) and the semantics must be inferred from the coordination structure.

Sentential gapping and similar constructions have been discussed before in the categorial grammar literature (Steedman, 1990). However, the goals of the previous literature was to assign grammatical structure to only those sentences that are grammatical and, as a result, resorted to increasing the class of grammars without accounting for the increased complexity in parsing such grammars. Our goal will be to provide the correct semantics within the framework of LCG.

The only mechanism that LCG allows to account for linguistic constructions is the lexicon. Therefore, we must determine which of the words in the sentence is the best functor in a sentential gapping construction. The two possible candidates are the verb in the left conjunct (the word "is" in (5.6)) and the coordinator itself (the word "and" in (5.6)). By choosing the latter, we assign special categories to the small set of possible

[8]Penn Treebank sentence 0998.35.

coordinators when they are used in a sentential gapping construction, as opposed to assigning special categories to the very large set of possible verbs. Then, the coordinator must assemble the meaning for the whole sentence from the other constituents. The correct semantics can be obtained by copying the verbal semantics from the verb in the left conjunct, and using them with the arguments of that verb in the right conjunct.

We will proceed with the syntactic and semantic structure for (5.6). To begin, we assume standard syntactic and semantic terms for words in the sentence[9] except the word "and":

Words	Syntactic Category	Semantic Term
Beacon Hill	NP	$bhill$
is	$S\backslash NP/PP$	$\lambda xy.(xy)$
for move-up town houses	PP	$\lambda x.for(x, muthouses)$
Nob Hill	NP	$nhill$
for single-family homes	PP	$\lambda x.for(x, sfhomes)$

According to this example lexicon, if the words in the left conjunct appeared in a sentence alone, they could form a sentence with the semantics $for(bhill, muthouses)$. However, the words in the right conjunct could not form a sentence by themselves. Therefore, our syntactic categories and semantic terms for the word "and" must combine these phrases to yield the correct syntax and semantics. To accomplish this, we add the following lexical entry for "and":

Word	Syntactic Category	Semantic Term
and	$S\backslash NP\backslash((S\backslash NP)/PP)\backslash PP/PP/NP$	$\lambda vwxyz.((yx)z) \wedge ((yw)v)$

This lexicon assigns categories to "and" that take five arguments corresponding to the five phrases discussed earlier. The word "is" is the fourth argument to "and", specified as

[9]To conserve space, we have assigned categories to phrases when the words within the phrase are not themselves interesting.

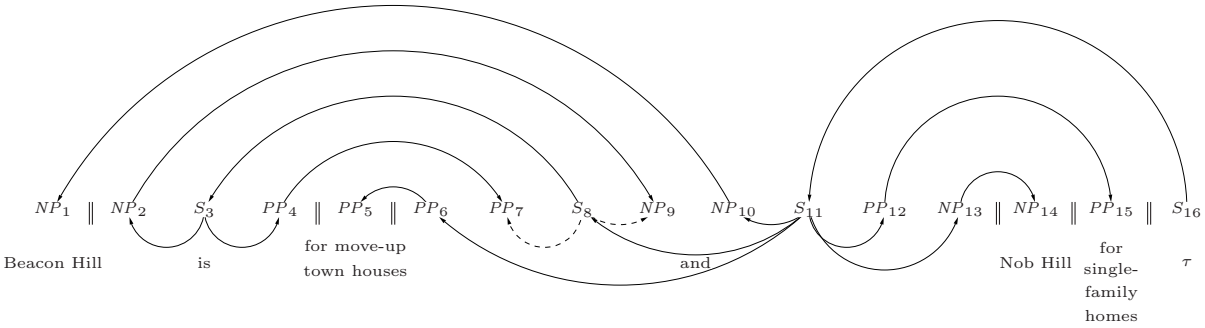


Figure 5.24: The LCG proof net for (5.6).

argument y, and can be seen to be copied and then applied to the appropriate arguments in the semantics for "and". The proof net for this sentence is shown in figure 5.24.

To obtain the semantic term for the proof net shown in figure 5.24, we simply apply the semantic term for "and" to the semantic terms for the various arguments in the syntactic derivation. We obtain the derivation in figure 5.25.

This analysis of sentential gapping has the primary advantage over previous proposals that it fits within a simple categorial grammar like LCG. In addition, this analysis does not require a large increase in the size of the lexicon, given that there are a small number of coordinators and a small number of categories for verbs in sentential gaps. Finally, it shows a high degree of transparency between the syntax and semantics of LCG, even with linguistic constructions that have caused problems for the semantics of other formalisms.

5.4 Converting CCGbank to an LCG corpus

In this section, we will specify our methods for translating CCGbank (Hockenmaier and Steedman, 2007) into an LCG corpus. CCGbank consists of approximately one million words of newspaper text drawn from the Wall Street Journal and divided into 48,934 sentences. The annotations of these sentences were semi-automatically translated from the phrase-structure trees of the Penn Treebank (Marcus et al., 1994). CCGbank contains

$$((((((\lambda vwxyz.((yx)z) \wedge ((yw)v))nhill)(\lambda x.for(x, sfhomes)))(\lambda x.for(x, muthouses)))(\lambda xy.(xy)))bhill)$$

$$\rightarrow (((((\lambda wxyz.((yx)z) \wedge ((yw)nhill))(\lambda x.for(x, sfhomes)))(\lambda x.for(x, muthouses)))(\lambda xy.(xy)))bhill)$$

$$\rightarrow ((((\lambda xyz.((yx)z) \wedge ((y(\lambda x.for(x, sfhomes))nhill)))(\lambda x.for(x, muthouses)))(\lambda xy.(xy)))bhill)$$

$$\rightarrow (((\lambda yz.((y(\lambda x.for(x, muthouses))z) \wedge ((y(\lambda x.for(x, sfhomes))nhill))))(\lambda xy.(xy)))bhill)$$

$$\rightarrow ((\lambda z.(((\lambda xy.(xy))(\lambda x.for(x, muthouses))z) \wedge (((\lambda xy.(xy))(\lambda x.for(x, sfhomes))nhill))))bhill)$$

$$\rightarrow ((\lambda z.(((\lambda xy.(xy))(\lambda x.for(x, muthouses))z) \wedge (((\lambda y.((\lambda x.for(x, sfhomes))y))nhill))))bhill)$$

$$\rightarrow ((\lambda z.(((\lambda xy.(xy))(\lambda x.for(x, muthouses))z) \wedge (((\lambda y.for(y, sfhomes))nhill))))bhill)$$

$$\rightarrow ((\lambda z.(((\lambda xy.(xy))(\lambda x.for(x, muthouses))z) \wedge for(nhill, sfhomes)))bhill)$$

$$\rightarrow (((\lambda xy.(xy))(\lambda x.for(x, muthouses))bhill) \wedge for(nhill, sfhomes))$$

$$\rightarrow ((\lambda y.((\lambda x.for(x, muthouses)y))bhill) \wedge for(nhill, sfhomes))$$

$$\rightarrow ((\lambda y.for(y, muthouses))bhill \wedge for(nhill, sfhomes))$$

$$\rightarrow (for(bhill, muthouses) \wedge for(nhill, sfhomes))$$

Figure 5.25: The LCG semantic derivation for (5.6).

CCG derivations, represented as phrase-structure trees, and dependency structures that can be derived from the CCG derivations.

We will present our method for semi-automatically translating the phrase-structure trees of CCGbank into LCG proof nets. The primary obstacle to this translation was the widespread presence of *non-categorial* rules in CCGbank, or rather, rules that do not correspond to any rules in the theoretical categorial grammar literature. Our general method was to lexicalize such rules by increasing the complexity of the lexicon to provide derivations in LCG. The two most widespread constructions that use non-categorial rules in CCGbank are relative clauses and coordination, for which we applied the solutions that

were presented in section 5.3.

Our translation was semi-automatic in the sense that we developed a framework for categorizing the rules of CCGbank, and then applied changes such as those discussed in section 5.3 to the phrase-structure trees. After this automatic step, approximately 500 rules were left without LCG derivations and we annotated these manually. It turned out that a great majority of these sentences were annotation errors in CCGbank.

In addition to the 48,934 sentences in CCGbank, 274 sentences that are present in the Penn Treebank were left out of CCGbank because they contain linguistic structures that were problematic for CCG. According to Hockenmaier (2003), these missing sentences each contain a *sentential gapping* construction, and are described in the following way by Hockenmaier (2003):

> This construction cannot be handled with the standard combinatory rules of CCG that are assumed for English.

We developed LCG derivations for these *left-out* sentences by training the Petrov PCFG parser on a phrase-structure representation of the LCG derivations that resulted from the semi-automatic translation of CCGbank. We then manually corrected the output of the Petrov PCFG parser to obtain LCG derivations for these sentences.

5.4.1 Lexicalizing the Crossing Rules of CCGbank

The crossing rules of CCG (see section 4.1) are not admissible in LCG and, therefore, we will need to find a way to lexicalize them. In section 4.1.3, we showed that there are two types of crossed composition rules in CCGbank: non-generalized crossed composition and reducing generalized crossed composition. In this section, we will specify methods of semi-automatically lexicalizing each of these two types.

Remember that the non-generalized crossed composition takes either a forward or a backward form:

- $X/Y, Y\backslash Z \rightarrow X\backslash Z$
- $Y/Z, X\backslash Y \rightarrow X/Z$

There are three properties of non-generalized crossed composition that allow us to easily lexicalize such rules. First, they are essentially all of the form $X/X, X\backslash X \rightarrow X\backslash X$ or $X/X, X\backslash X \rightarrow X/X$. Second, X is nearly always the category $S\backslash NP$. Third, the two child nodes are nearly always leaf nodes.

Because of these three properties, we can see that non-generalized crossed composition is being used to specify the direction of application between some kind of verb phrase and its modifier. However, we can simply change the direction of one of the slashes and alter the semantic counterpart to exchange the order of application and we obtain an instance of non-crossed composition.

Reducing generalized crossed composition rules in CCG take one of the following two forms:

- $X/X, X|_1 Z_1|_2 \ldots|_n Z_n \rightarrow X|_1 Z_1|_2 \ldots|_n Z_n$
- $X|_1 Z_1|_2 \ldots|_n Z_n, X\backslash X \rightarrow X|_1 Z_1|_2 \ldots|_n Z_n$

Such rules specify the combination of a category $X|_1 Z_1|_2 \ldots|_n Z_n$ with a modifier category X/X or $X\backslash X$. In nearly all cases, the position of the category $X|X$ is at a leaf node. In both the forward and backward cases, the category $X|X$ can be rewritten as $X|_1 Z_1|_2 \ldots|_n Z_n|(X|_1 Z_1|_2 \ldots|_n Z_n)$. Then, to maintain the same semantics, the semantic term of the word needs to be changed to apply its arguments to the semantic counterpart of the category $X|_1 Z_1|_2 \ldots|_n Z_n$ first. This increases the size of the lexicon by a factor, but only for the categories participating in generalized crossed composition, of which there are a small number.

5.4.2 Semi-automatic Translation Framework

CCGbank consists of approximately 1,000,000 words of text and, due to the binary-branching nature of CCG, there are approximately 1,000,000 phrase-structure rules specifying the phrase-structure trees for those sentences. Approximately 100,000 of the phrase-structure rules found in CCGbank are non-categorial, and many of the *categorial* rules are directly translatable into LCG derivations. Section 5.3 showed the primary linguistic constructions that use non-categorial rules. Automatically applying the solutions described in that section provided LCG derivations for nearly all of the non-categorial rules in CCGbank.

Before we could apply such automatic translations to the rules of CCGbank, we needed to identify which rules correspond to which linguistic constructions. We developed a Java application for visualizing the phrase-structure trees of CCGbank and for developing a hierarchy of rule types. Using this hierarchy of rule types, we specified a translation to eliminate the non-categorial rules in the phrase-structure tree. Once this was accomplished, only a few hundred non-categorial rules remained, which were annotated manually during the hierarchy building process.

The crossed rules of CCGbank are not compatible with the definition of LCG as we have defined it in this dissertation. Therefore, to obtain LCG rules, we lexicalized the crossed rules of CCGbank as discussed in section 5.4.1. However, categorial grammar formalisms that can be represented as proof nets and, that allow crossed rules, have been discussed in the literature (Buch, 2009). Therefore, in developing our LCG derivations, we eliminated the crossed rules in the last step of the translation process, and also retained a version of our LCG derivations that used the crossed rules.

5.4.3 Annotating the Left-Out Sentences

In our approach for annotating the 275 left-out sentences with LCG derivations, we trained the Petrov parser (Petrov et al., 2006) on the LCG derivations for the sentences in CCGbank. Then, we proceeded with a manual re-annotation for the problematic constructions in those sentences. The Petrov parser requires phrase-structure derivations as input, so we used the phrase-structure representation of our LCG derivations.

5.4.4 Proof Nets as XML

One of the primary advantages of using LCG over CCG is the fact that we can capture both the phrase-structure trees and the dependency structures of CCGbank by using the single structure of proof nets. Due to this fact, our corpus of LCG derivations is represented as proof nets. We have chosen the XML file format, due to the difficulty that we had working with various other file formats in the various corpora that we used in developing this corpus.

The XML representation of the LCG proof net in figure 5.3 is shown in figure 5.26. The category for the whole sentence is given as tag "sentential", then the lexicon is given and finally the proof net is given. Punctuation such as "," and "." is included in the lexicon, but excluded from the proof net.

5.5 Evaluation of LCGbank

In this section, we evaluate LCGbank, whose construction was described in section 5.4. The existing literature has evaluated highly-lexicalized corpora in various ways and we will use three of these methods here. First, corpora have been evaluated via a statistical analysis of the contents of the corpora, including statistics such as average number of categories per word and coverage differences between train and test sets (Hockenmaier, 2003, Hockenmaier and Steedman, 2007). Second, corpora have been evaluated via the

```
<sentence id="0351.4">
        <sentential cat="S_29"/>
        <words>
                <word text="The" cat="NP_0/N_1"/>
                <word text="market" cat="N_2"/>
                <word text="'s" cat="NP_3/N_4\NP_5"/>
                <word text="tempo" cat="N_6"/>
                <word text="was" cat="S_7\NP_8/(S_9\NP_10)"/>
                <word text="helped" cat="S_11\NP_12"/>
                <word text="by" cat="S_13\NP_14\(S_15\NP_16)/NP_17"/>
                <word text="the" cat="NP_18/N_19"/>
                <word text="dollar" cat="N_20"/>
                <word text="'s" cat="NP_21/N_22\NP_23"/>
                <word text="resiliency" cat="N_24"/>
                <word text=","/>
                <word text="he" cat="NP_25"/>
                <word text="said" cat="S_26\S_27\NP_28"/>
                <word text="."/>
        </words>
        <matching>
                <match first="1" second="2"/>
                <match first="0" second="5"/>
                <match first="4" second="6"/>
                <match first="19" second="20"/>
                <match first="18" second="23"/>
                <match first="22" second="24"/>
                <match first="17" second="21"/>
                <match first="12" second="16"/>
                <match first="11" second="15"/>
                <match first="10" second="14"/>
                <match first="9" second="13"/>
                <match first="3" second="8"/>
                <match first="25" second="28"/>
                <match first="7" second="27"/>
                <match first="26" second="29"/>
        </matching>
</sentence>
```

Figure 5.26: The XML representation of the LCG proof net for sentence 0351.4.

performance of a supertagger on their lexicons (Clark and Curran, 2007b, Moot, 2010b,a, 2015). Finally, corpora have been evaluated via the performance of wide-coverage statistical parsers (Charniak, 2000, Hockenmaier, 2003, Clark and Curran, 2007b).

Throughout this section, we will evaluate two different versions of LCGbank: LCGbank without the left-out sentences from CCGbank (referred to as LCGbank w/o left-out) and LCGbank including the left-out sections (referred to as LCGbank or LCGbank proper). The former gives a good comparison point with CCGbank, due to the exclusion of the left-out sentences, and the latter gives a good comparison point with the Penn Treebank, due to their inclusion. In section 5.5.1, we perform a statistical analysis of the lexicon of CCGbank. Hockenmaier and Steedman (2007) also included a coverage analysis, which we will omit here because our coverage is complete, and a syntactic analysis, which we will omit here because LCG is entirely lexicalized. In section 5.5.2, we train a supertagger on LCGbank and evaluate it, comparing results to Clark and Curran (2007b). Finally, in section 5.5.3, we train the Petrov parser on the phrase-structure representation LCGbank and evaluate it, comparing results to those of chapter 4.

The evaluation in this section is fundamentally different from the evaluation of a parser such as that in chapter 4, because the goal of a corpus is to represent the structure of language, rather than reproduce a held-out section. Because of this, lower or higher numbers do not necessarily represent superiority or inferiority, but instead, a difference in the structure of the corpus.

5.5.1 Evaluation of the Lexicon

Table 5.6 shows some basic statistics on the words and categories in the lexicons of CCGbank, LCGbank without the left-out sentences and LCGbank proper. The average categories per word is simply the number of unique word-category pairs divided by the number of unique words, and the expected categories per word is the weighted average according to frequency. The degree of lexical ambiguity in LCGbank is higher than

	CCGbank	*LCGbank w/o left-out*	*LCGbank*
Word-category pairs	74669	85245	85720
Unique words	44210	44210	44387
Total words	929552	929552	936022
Average categories per word	1.689	1.928	1.931
Expected categories per word	19.186	26.321	28.585
Average order of categories	0.785	0.838	0.838

Table 5.6: Basic statistics for sections 02-21.

that of CCGbank, which is expected due to the lexicalization of the non-categorial and crossing rules in CCGbank.

Tables 5.7a and 5.7b show the most frequently occurring words, along with the counts of their categories and their frequencies. In comparison to CCGbank, LCGbank without the left-out sentences has much higher degree of ambiguity for conjunctions such as "and", "but" and ",", because they have a full categorial treatment in LCGbank, and a much lower degree of ambiguity for modal verbs such as "is", because of the lack of features in LCGbank. Comparing LCGbank without the left-out sentences and LCGbank proper, the conjunctions "and" and "," have a much higher degree of ambiguity because of their grammatical function within the sentential gapping constructions common in the left-out sentences.

Figures 5.27 and 5.28 show the growth of the total number of categories in the lexicon as we add categories for an increasingly large number of words in the corpus. Thresholds are applied for $x = 0, 1, 2, 4$ where we ignore categories that have appear x or less times. So, $f > 0$ is the size of the entire lexicon for each point in the word sequence. These figures give us an indication of how likely we are to encounter words in unseen data for which

Word	*# of Categories*	*Frequency*
and	126	15906
as	122	4237
in	120	15085
to	109	22056
,	81	48310
not	81	1288
up	69	1683
than	69	1600
for	68	7912
at	67	4313
on	66	5112
of	63	22782
or	63	2489
is	57	6893
before	56	519
from	55	4437
out	55	1030
after	55	926
only	51	928
but	50	1641

(a) LCGbank w/o left-out.

Word	*# of Categories*	*Frequency*
and	170	16115
as	125	4256
in	121	15186
to	115	22198
,	103	48723
not	81	1300
than	71	1606
for	69	7976
on	69	5145
up	69	1698
at	68	4362
of	65	22929
or	63	2500
is	58	6938
from	56	4459
before	56	524
out	55	1036
after	55	930
but	52	1662
only	52	938

(b) LCGbank.

Table 5.7: The 20 words with the highest number of lexical categories and their frequency (sections 02-21).

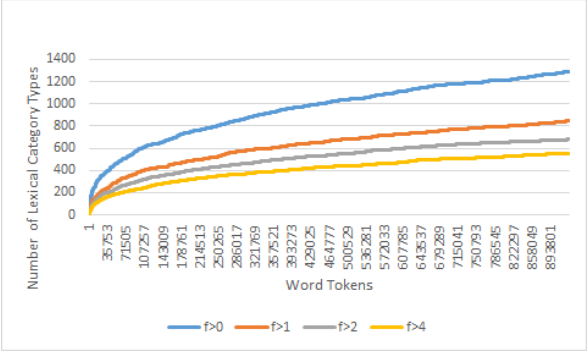
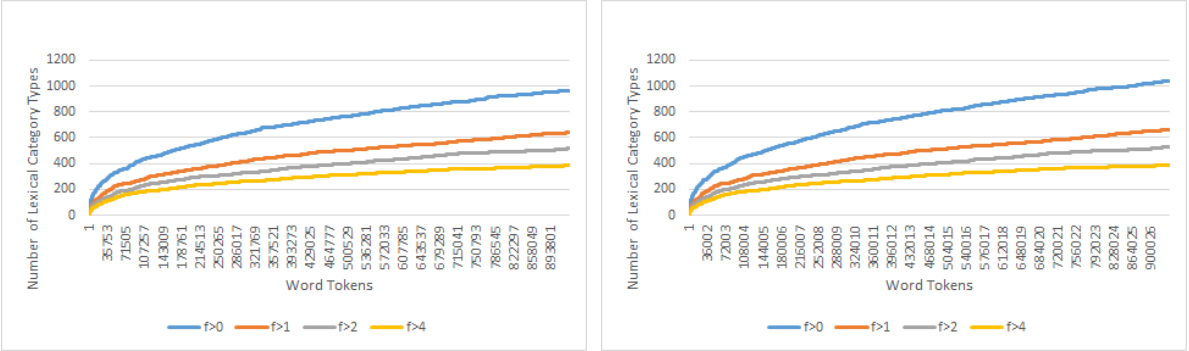


Figure 5.27: The growth of the lexicon of CCGbank in sections 02-21.



(a) LCGbank w/o left-out.

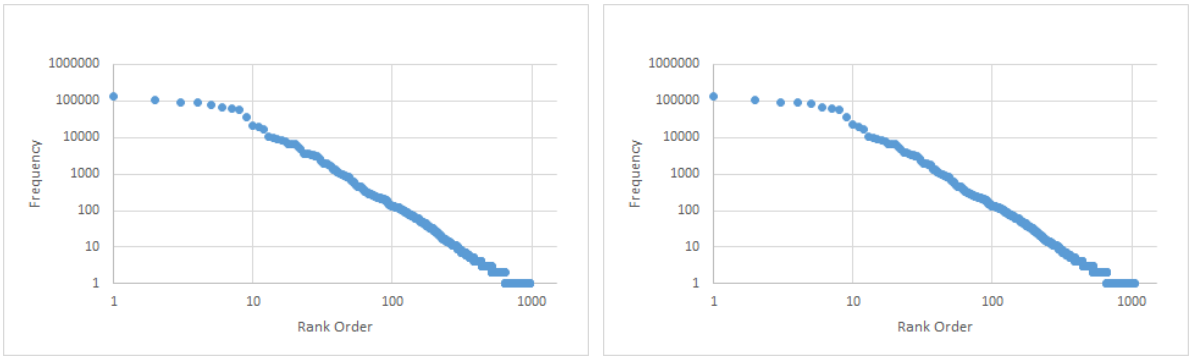
(b) LCGbank.

Figure 5.28: The growth of the lexicon of LCGbank in sections 02-21.

the correct category is not yet known. In particular, they indicate that the behaviour of the three corpora are nearly identical in this regard. In contrast to Hockenmaier and Steedman (2007), we did not find that there is a point where the size of the lexicon stops growing for the higher thresholds on any of the three corpora.

Figure 5.30 shows log-log plots of the frequency of the category type against its rank. Because the relationship between the two variables is linear on this log-log plot, this is an indication that the distribution of category frequencies follows a Zipfian distribution.

Table 5.8 gives some statistics on section 00 based on the lexicon from sections 02-21. This gives an indication of the source of unknown information across the three corpora.



(a) LCGbank w/o left-out.

(b) LCGbank.

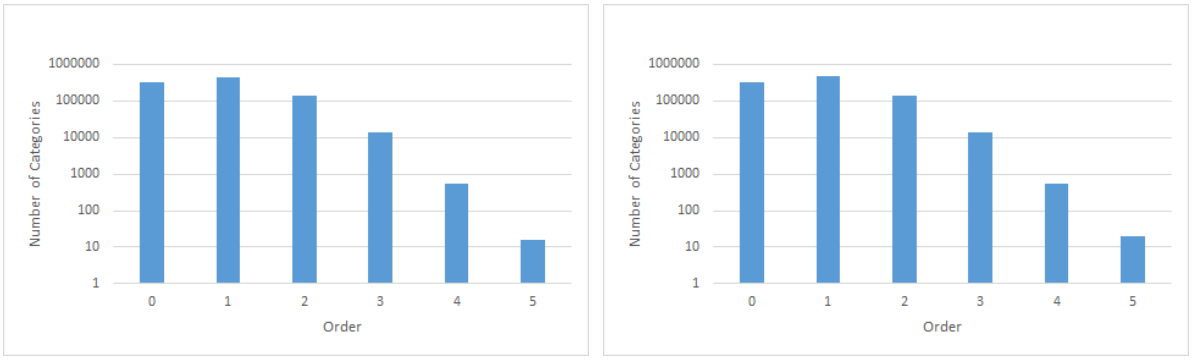
Figure 5.29: A log-log plot of the rank order and frequency of the lexical category types in LCGbank.

	CCGbank	*LCGbank w/o left-out*	*LCGbank*
Cats in section 00 found in sections 02-21	42707	42300	42528
Total cats in section 00	45422	45422	45656
Percentage found	94.02	93.13	93.15
Percentage completely unknown words	3.80	3.80	3.80
Known words but unknown word-cat pair	2.17	3.07	3.05

Table 5.8: Statistics on unseen data in section 00.

As before, the increased lexical ambiguity of LCGbank causes an increase in the number of unknown word-category pairs in section 00, relative to CCGbank.

Finally, figure 5.30 shows the order of LCGbank, which can be contrasted with figure 3.29 in chapter 3. There is a slight increase in the number of categories of order 5 when the left-out sentences are added to LCGbank, but otherwise, the distribution of order across the three corpora are nearly identical.



(a) LCGbank w/o left-out.

(b) LCGbank.

Figure 5.30: The order of categories in LCGbank in sections 02-21.

5.5.2 Supertaggers on LCGbank

In this section, we evaluate LCGbank by training a supertagger on it and reporting the results. This will give an indication as to how difficult it will be to learn the increased lexical ambiguity of LCGbank.

We train Clark and Curran's maximum entropy supertagger (Clark and Curran, 2007b) on LCGbank and compare those results with the same system trained on CCGbank. To compare our results to those of Clark and Curran (2007b), we train on sections 02-21 and test on section 00. Tables 5.9 and 5.10 give the supertagging ambiguity accuracy on LCGbank without the left-out sentences and on LCGbank proper, respectively, in the same format as Table 4 from Clark and Curran (2007b).

The β column restricts the supertagged categories to only those with a probability within a factor of β of the highest probability category. The (G) columns use gold standard part-of-speech tags as part of the training data and the (A) columns use automatically generated part-of-speech tags.

The results indicate that supertagging is generally a little more difficult on LCGbank without the left-out sentences than it is on CCGbank, with the notable exception of the lowest value of β. Relative to LCGbank without the left-out sentences, the results on

β	k	*CATS/WORD*	*ACC (G)*	*SENT ACC (G)*	*ACC (A)*	*SENT ACC (A)*
0.075	20	1.35	96.27	63.36	95.42	59.38
0.030	20	1.56	97.24	69.89	96.49	65.66
0.010	20	1.96	97.80	74.23	97.17	69.99
0.005	20	2.33	98.05	75.69	97.51	71.82
0.001	150	4.32	99.10	86.41	98.71	81.86

Table 5.9: Supertagger ambiguity and accuracy on section 00 for LCGbank w/o left-out.

β	k	*CATS/WORD*	*ACC (G)*	*SENT ACC (G)*	*ACC (A)*	*SENT ACC (A)*
0.075	20	1.35	96.26	63.46	95.43	59.60
0.030	20	1.57	97.22	69.76	96.44	65.43
0.010	20	1.98	97.80	74.13	97.15	69.96
0.005	20	2.35	98.05	75.79	97.50	71.89
0.001	150	4.38	99.10	86.41	98.72	82.09

Table 5.10: Supertagger ambiguity and accuracy on section 00 for LCGbank.

LCGbank proper are very similar, which indicates that the additional lexical ambiguity due to the categories for sentential gapping is not problematic.

5.5.3 The Petrov Parser on LCGbank

Our final method for evaluating LCGbank is to train the Petrov parser on it, and compare its performance to the Petrov parser on other corpora from chapter 4. This is possible because, although LCGbank is a corpus of LCG derivations, it was converted from CCGbank and it was only in the last step that the conversion from the phrase-structure of CCGbank to the proof nets of LCG was done. Furthermore, the evaluation

	Labelled %			*Unlabelled %*			
Parser	*R*	*P*	F	*R*	*P*	*F*	*Coverage*
I-1	76.72	76.76	76.74	82.49	82.53	82.51	99.84
I-4	83.99	84.01	84.00	88.82	88.83	88.82	99.84
I-5	84.47	84.49	84.48	89.28	89.30	89.29	99.84
I-6	84.67	84.69	84.68	89.41	89.43	89.42	99.84

Table 5.11: Constituent accuracy for the Petrov parser on LCGbank w/o left-out on section 00.

	Labelled %			*Unlabelled %*			
Parser	*R*	*P*	F	*R*	*P*	*F*	*Coverage*
I-1	77.00	77.02	77.01	82.76	82.78	82.77	99.90
I-4	83.81	83.82	83.82	88.67	88.69	88.68	99.90
I-5	84.65	84.66	84.66	89.35	89.36	89.35	99.90
I-6	84.78	84.78	84.78	89.43	89.44	89.43	99.90

Table 5.12: Constituent accuracy for the Petrov parser on LCGbank on section 00.

in this section will be a constituent analysis only. An analysis involving a conversion of the phrase-structure output by the Petrov parser to proof nets for a proof net evaluation is left to future research.

Tables 5.11 and 5.12 give the performance of the Petrov parser after various iterations for both LCGbank without the left-out sentences and for LCGbank proper evaluated on section 00. Then, table 5.13 gives the performance of the best performing models on section 23 and contrasts them with the other constituency results from chapter 4.

The accuracy on LCGbank is somewhat lower than the accuracies on the Penn Tree-

	Labelled %			*Unlabelled %*			
Parser	*R*	*P*	F	*R*	*P*	*F*	*Coverage*
Petrov on Penn Treebank I-6	89.65	89.97	89.81	90.80	91.13	90.96	100.00
Petrov on CCGbank I-5	86.94	86.80	86.87	90.75	90.59	90.67	99.83
Petrov on CCGbank no feats I-6	87.49	87.49	87.49	90.81	90.82	90.81	99.96
Petrov on LCGbank w/o left-out I-6	85.72	85.74	85.73	90.03	90.06	90.05	99.92
Petrov on LCGbank I-6	85.57	85.59	85.58	89.92	89.95	89.93	99.79

Table 5.13: Constituent accuracy for the Petrov parser on the corpora including LCGbank on all sentences from section 23.

bank and CCGbank, which is an indication that the syntactic structure in LCGbank is more difficult to learn. However, this is an expected trade-off of having more of the semantics represented in the syntax.

5.6 LCG Parsing in Practice

In chapter 3, we introduced a parsing algorithm for LCG with a worst case complexity of $O(n^4)$ when order is bounded by a constant, and $O(n^3)$ when lexical ambiguity and order is bounded by a constant. Despite a polynomial running time, there are still two potential causes of slowness for the algorithm. First, the constants in the proof are exponential in the order of the input categories k, specifically $2^{2^{2k}}$ and 2^{10k}. Second, if n is high and the algorithm runs in $O(n^3)$ or $O(n^4)$ in practice, the algorithm will be prohibitively slow. In this section, we will demonstrate that neither of these potential causes of slowness are present, and that the algorithm is efficient in practice.

To demonstrate the efficiency of our algorithm in practice, we developed a Java im-

plementation[10] of our proof net parser from chapter 3. Then, we ran the algorithm on our corpus of LCG derivations from earlier in this chapter. The experiments were done on a machine with 4GB of RAM and an Intel Core i7-2677M CPU at 1.8Ghz. As with the experiments in the CCG literature, the lexicon is built from sections $02-21$, and we ran the algorithm on the sentences from section 00. For unknown words, we assigned the two syntactic categories N and NP in the lexicon.

As in the CCG literature, we did not use the complete set of categories assigned to words in sections $02-21$ in the lexicon. Instead, we applied a threshold to the frequencies of the occurrences of a category, relative to the total number of occurrences of a given word. We used three thresholds: 0.1, 0.01 and 0.005. These three thresholds yielded three distinct sets of inputs to the LCG problem, each with an increasing average number of categories per word. Since we are only interested in the efficiency of the algorithm, we will treat all of these inputs as a single large set of inputs. This is not an unreasonable representation of the parsing space that would be seen by a more sophisticated parser, due to the widespread use of supertagging in the CCG literature (Clark and Curran, 2007b). We ran the algorithm on these inputs, and recorded the number of milliseconds on each input and the number of ATGs in the resultant chart.

Figure 5.31 shows both the number of graphs charted against the number of atoms and the number of milliseconds charted against the number of atoms. That figure also shows polynomials fitted to the data using the least squares method. We show the polynomial of highest degree that did not overfit the data, according to a visual inspection. The fitted polynomial for the number of milliseconds against the number of atoms is $y = 0.0054x^2+5.2399x-1457.4$ and the fitted polynomial for the number of graphs against the number of atoms is $y = -3*10^{-13}x^5+5*10^{-9}x^4-8*10(-5)x^3+0.0591x^2-15.261x+1105.6$.

Figure 5.32 shows the number of graphs charted against the number of milliseconds

[10]This implementation should be considered research-quality code, as it has not been tuned for optimal efficiency.

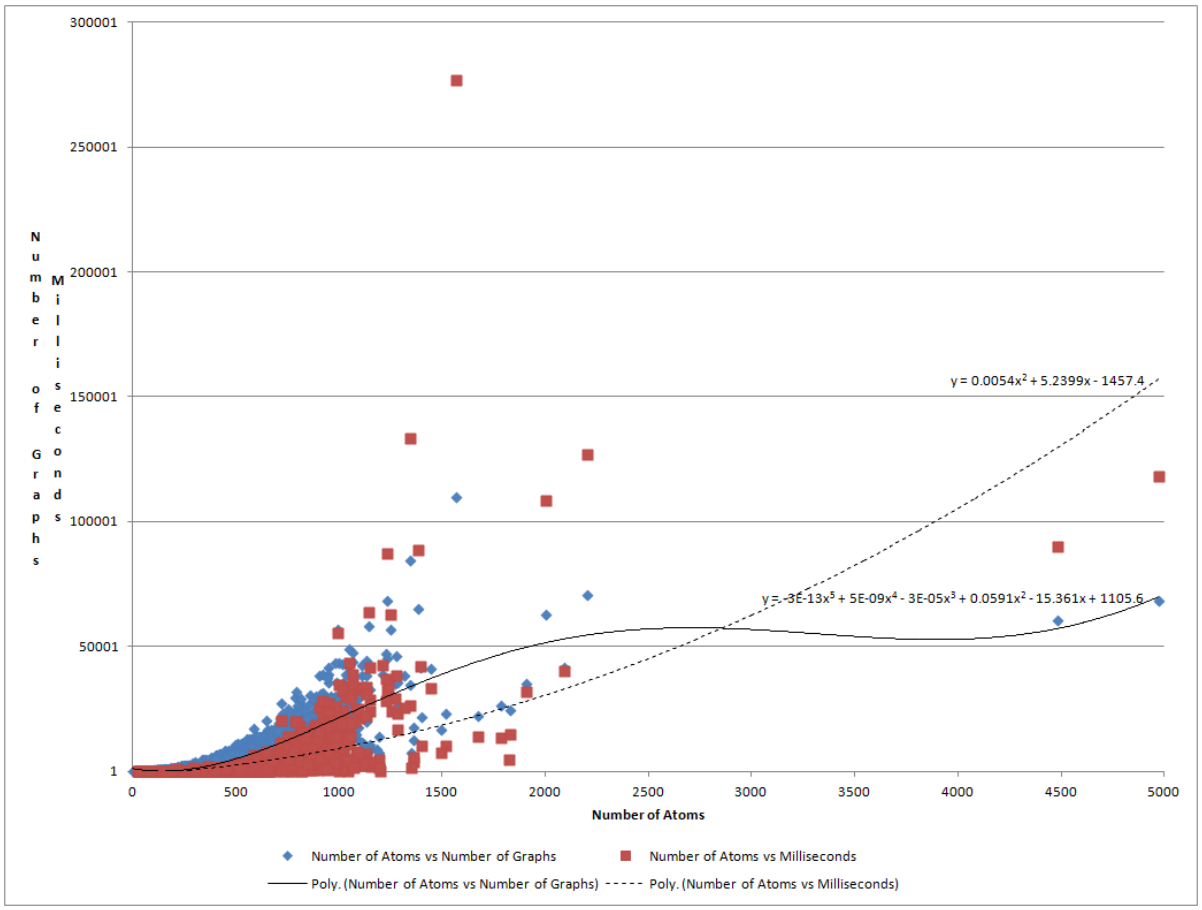


Figure 5.31: Number of Atoms vs Milliseconds and Number of Graphs.

for each input. A polynomial fitted to the data using least squares is also shown and it is $y = 0.0001x^2 - 0.0481x - 70.08$.

The proofs of running time in chapter 3 showed that our algorithm runs in time $O(n^4)$ in the worst case. Closer examination of those proofs yield that n^2 of that time is due to there being a possibly quadratic number of graphs in the chart, and the remaining n^2 is due to having to adjoin to a possibly quadratic number of graphs for each graph in the chart. In the remainder of this section, we will discuss how the data demonstrates that, although there are on average a quadratic number of graphs in each chart, processing each graph is very close to constant time, rather than quadratic.

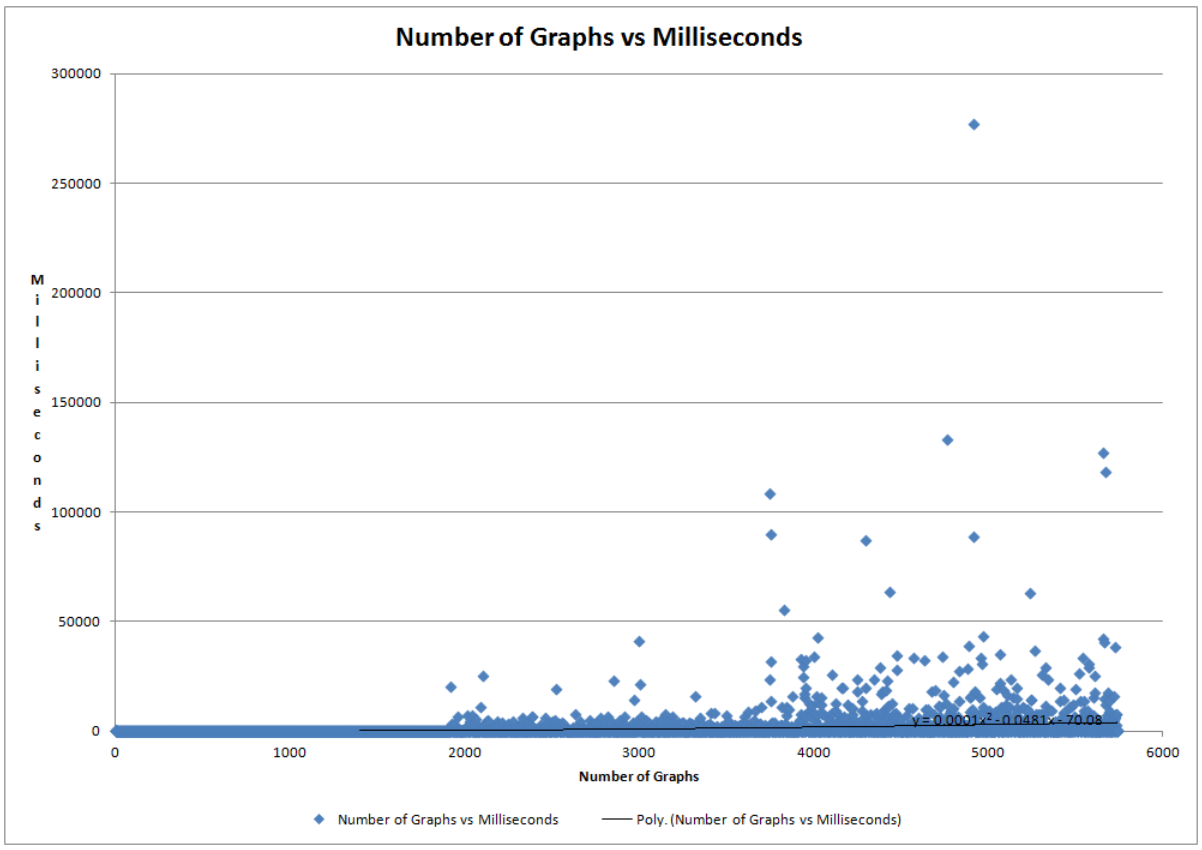


Figure 5.32: Number of Graphs vs Number of Milliseconds.

To understand the average running time of the algorithm on these inputs, we should look to the polynomials of best-fit in figure 5.31. The polynomial for the number of milliseconds indicates that this algorithm runs in quadratic time on average, with a small constant, as opposed to the possibly huge constants from the proof in chapter 3. The polynomial for the number of graphs per atom indicates that the reason for this is that there are, on average, a quadratic number of graphs per atom.

Figure 5.32 demonstrates even further that the processing of graphs is happening in close to constant time, as opposed to the provably worst-case quadratic time from chapter 3. The best-fit polynomial is quadratic, but with a very low constant, indicating that in practice, the number of adjoinings of graphs done during chart parsing is very small.

Finally, we discuss the average time that our parser took relative to other state-of-the-art parsers. On average, our parser took 160 milliseconds per sentence with the threshold of 0.01, 823 milliseconds per sentence with the threshold of 0.001 and 2,842 milliseconds per sentence with the threshold of 0.005, for a total of 1,224 milliseconds per sentence for the whole data set. In contrast, the Clark and Curran parser took 81 milliseconds per sentence, the Charniak parser took 1,194 milliseconds per second and the Collins parser took 1,918 milliseconds per sentence (Clark and Curran, 2007b).

While our results are slower than the current-state-of-the-art parsers, we can see that with the threshold of 0.01, which is using all of those categories that occur more than 1% of the time for a given word, we get fairly competitive parsing times. With supertagging, a probabilistic model to prune the search space and some effort put into tuning the code, our algorithm should be competitive with the state of the art.

Chapter 6

Conclusion

The primary subject of this dissertation is the bridging of the gap between the heretofore theoretical realm of Lambek Categorial Grammar (LCG) and the primarily practical realm of Combinatory Categorial Grammar (CCG). The primary arguments have been for the use of LCG over CCG, although it may be the case that some of the notions that we have developed in this dissertation may also be beneficial to the CCG community. Our contributions have been both theoretical and practical, although always with an eye towards the development of a practical LCG parser. Thus, we developed a number of results that supported our thesis statement that LCG is viable for practical parsing and that its syntax-semantics interface has greater transparency than that of CCG.

6.1 Overview

Our first topic of focus towards developing a practical LCG parser was to develop an LCG parsing algorithm. The NP-completeness of the LCG parsing problem has generally been considered too prohibitive for practical use. The lack of interest in developing an efficient LCG parsing problem is in spite of the existence of proof nets (a very natural and efficient representation of a parse in LCG), and the extensive literature on proof nets for a variety of grammar formalisms. Chapter 3 reveals that proof nets are a very concise

and efficient representation of parses for LCG in particular. In addition to providing an efficient parsing algorithm for LCG, that chapter paves the way for research into parsing algorithms based on proof nets for similar categorial grammar formalisms. Furthermore, we can see that the proof nets that we developed in chapter 3 are fundamental to the structure of LCG because they are nearly identical to the proof net representation that was used to prove the NP-completeness of the LCG parsing problem (Savateev, 2008).

In addition to concerns about efficiency, the parsing and grammar literature is generally concerned about the generative capacity of the grammar formalism that is used as the basis of a parser. The concern over the weak generative capacity of parsing formalisms has led to the development of a number of non-context-free grammar formalisms, such as CCG, Tree-Adjoining Grammars, Linear-Indexed Grammars and Head Grammars, among many others. The concern is well-founded, if we are interested in modelling the grammatical structure of all languages, due to the existence of cross-serial dependencies in languages such as Swiss German (Shieber, 1985) and Bambara (Culy, 1985). However, what has been lost in the drive towards mild context-sensitivity (Kallmeyer, 2013, Kuhlmann et al., 2015) is the extent to which it actually occurs in our parsing data and the extent to which our parsers take advantage of the mild context-sensitivity of their underlying grammar formalisms. In chapter 4, we performed an analysis of the CCGbank (Hockenmaier and Steedman, 2007) literature, and, in particular, the Clark and Curran parser (Clark and Curran, 2007b), to determine the effect of the mild context-sensitivity of CCG on that parser's performance. The result is quite surprising, insofar as the Clark and Curran parser and CCGbank obtain no benefit whatsoever from the mild context-sensitivity of CCG.

These two topics, together, lead us to the conclusion that the grammar and parsing research communities have been blinded by the high-level properties of grammar formalisms. Or, in other words, the complexity of the parsing problem and the weak generative capacity of the grammar formalism have guided research, whereas the low-level

details tell an entirely different story. It is our feeling that this disconnect is a direct result of the gap between the theoretical grammar literature and the practical parsing literature, and that closing that gap will yield a much better understanding of exactly what is happening in the practical parsing literature. One way to view our research in chapters 3 and 4 is that it is making steps towards the goal of better understanding parsing formalisms in practice. That is, we have reconsidered widely-assumed truisms in the literature, such as NP-completeness being too inefficient in practice and weak context-freeness being too weak in practice, and doing so has led to fruitful new areas of research.

Another major contribution of this dissertation is in establishing the extent of the underlying differences between Context-Free Grammars (CFGs) and CCG, especially the CCG of the Clark and Curran parser. There are some theoretical differences between the two formalisms, such as their weak generative capacity and the complexity of their parsing problems, which are fundamental theoretical differences. However, in chapter 4 we established that the practical implementations of CCG are *strongly* equivalent to CFGs. We further demonstrated their similarity by training a probabilistic CFG parser on the CCG training corpus to demonstrate that these theoretical differences are not the source of the advantages of the Clark and Curran parser. This leads us to an important conclusion, which is that the major difference between the so-called CFG parsers and the Clark and Curran parser is not theoretical properties of their grammars, but rather the specific choices in the terminals and non-terminals that are used in the grammar. In particular, the choices of non-terminals in categorial grammar allow a semantic grammar to be built alongside the syntactic one, which in turn allows for the generation of semantic terms. In other words, it is the usage of the particular terminals and non-terminals in the Penn Treebank that cause problems with semantics, not the inability of CFGs to model infinitely long cross-serial dependencies.

In chapter 5, we developed a corpus of LCG derivations from CCGbank, which has

a number of important consequences. This is the first LCG corpus as well as the first proof net corpus, which opens the door for a variety of new research opportunities.

Because we now have an LCG corpus, as well as now having an efficient LCG parsing algorithm, we can begin the development of a practical parser that uses LCG as its grammar formalism. This means that we can begin to explore the advantages of LCG over other formalisms that we outlined in chapter 5. In particular, we can take advantage of the full associativity of LCG, and its effect on semantics and on normal forms of parses. Furthermore, we can begin to explore the effect of the close relationship between dependency structures and proof nets on practical parsing. In particular, developing a probability model for proof nets should be straightforward, using the probability model for CCG dependency structures developed for the Clark and Curran parser as a starting point (Clark and Curran, 2007b).

The development of the first corpus of proof nets in chapter 5 also allows us to explore the use of parsers based on proof nets for other categorial grammar formalisms. The proof net literature is very extensive and has included the development of proof nets for other variants of LCG (Fowler, 2007, Buch, 2009), other variants of categorial grammar (Moortgat, 2007) and linear logic (Lamarche, 2008). Having a corpus of LCG proof nets allows proponents of other categorial grammar formalisms to identify specific strengths of their formalism on actual sentences that already have an LCG proof net for comparison. We expect that this will lead to more interest in proof nets from the practical parsing community as a greater number of categorial grammars can now be used for the basis of practical parsers.

6.2 Future Directions

Having made these arguments for both the viability of LCG and the advantages of LCG, the next step is to build a practical parser based on LCG. Further steps would be to

develop formal proofs for the parts of chapter 3 that take the form of intuitive arguments here. Then, one could build a non-probabilistic parser based on the research in this dissertation. However, comparing the performance of such a parser to the state of the art would be unfair. This unfairness is due to probabilistic parsers offering accuracy advantages by aiding ambiguity resolution, and offering speed advantages by allowing pruning of the parse space.

To build a probabilistic parser using LCG, research must be done to develop probability models for LCG proof nets. A precedent for such models exists in the work of Clark and Curran (2007b) on dependency models for categorial grammar parsing, which will likely be adaptable to probabilistic models over proof nets.

However, if we were to build a probabilistic LCG parser, there are issues with the existing evaluation methods of parsers. The current literature uses three types of evaluation for parsers based on categorial grammar. First, they have been evaluated by comparing the dependencies that they produce for a held-out test set against those found in the corpus (Clark and Curran, 2007b). This method makes it difficult to compare parsers built on different grammar formalisms because the particular dependencies represented in a corpus vary quite a bit among the various corpora. Second, the coverage of the semantic component of the parser can be evaluated (Bos et al., 2004), or rather, one can simply calculate the number of sentences for which a semantic term can be constructed. However, this tells us nothing about the quality of the semantics, which makes such an evaluation extremely limited. Finally, we can evaluate the performance of a parser by embedding it in a semantic task (Bos and Markert, 2005). The biggest challenge with this approach is in isolating the parser from the rest of the components in the system, since these systems involve a number of interrelated components, including systems such as knowledge bases and inference systems. Therefore, an avenue of future research is to investigate better evaluation methods for parsers.

Another promising avenue of research is to generalize and expand the LCG parsing

algorithm of chapter 3. The parsing algorithm in chapter 3 is a parsing algorithm for LCG without products and allowing empty premises. Generalizing the algorithm to include products and include the case where premises are banned would bring the algorithm more in line with the theoretical LCG literature (Lambek, 1958, Fowler, 2006). Such research can continue down the trail of categorial grammars by extending this algorithm to other related categorial grammar formalisms, such as the Lambek-Grishin calculus (Moortgat, 2007), which may yield insight into the complexity of their parsing problems.

Bibliography

E. Aarts. Proving theorems of the second order Lambek calculus in polynomial time. *Studia Logica*, 53(3):373–387, 1994.

E. Aarts and K. Trautwein. Non-associative Lambek categorial grammar in polynomial time. *Mathematical Logic Quarterly*, 41(4):476–484, 1995.

K. Ajdukiewicz. Die syntaktische konnexität. *Studia Philosophica*, 1:1–27, 1935. McCall, S. (Ed. trans.) (1967). Polish Logic: 1920–1939. Oxford: Oxford University Press, page 207-231.

M. Auli and A. Lopez. 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*, page 470–480, Portland, Oregon, 2011a.

M. Auli and A. Lopez. Training a log-linear parser with loss functions via softmax-margin. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, page 333–343, Edinburgh, Scotland, UK, 2011b.

J. Baldridge. *Lexically specified derivational control in Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh, 2002.

R. Bar-Haim, I. Dagan, B. Dolan, L. Ferro, D. Giampiccolo, B. Magnini, and I. Szpektor. The second PASCAL Recognising Textual Entailment challenge. In *Proceedings of the*

Second PASCAL Challenges Workshop on Recognising Textual Entailment, page 1–9, 2006.

Y. Bar-Hillel. A quasi-arithmetical notation for syntactic description. *Language*, 29(1): 47–58, 1953.

Y. Bar-Hillel, C. Gaifman, and E. Shamir. On categorial and phrase-structure grammars. *Bulletin of the Research Council of Israel*, 9F:1–16, 1960.

J. Bos and K. Markert. Recognising textual entailment with logical inference. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, page 628–635, Vancouver, Canada, 2005.

J. Bos, S. Clark, M. Steedman, J. R. Curran, and J. Hockenmaier. Wide-coverage semantic representations from a CCG parser. In *Proceedings of the 20th International Conference on Computational Linguistics*, page 1240–1246, Geneva, Switzerland, 2004.

S. Boxwell. The PARG generator, 2010. http://www.ling.ohio-state.edu/~boxwell/software/parg_generator.html.

A. Buch. Mildly non-planar proof nets for CCG. In *Proceedings of the European Summer School for Logic, Language, and Information Student Session*, page 160–169, Bordeaux, France, 2009.

B. Carpenter. *Type-logical semantics*. The MIT Press, 1998.

B. Carpenter and G. Morrill. Switch graphs for parsing type logical grammars. In *Proceedings of the 9th International Workshop on Parsing Technology*, page 18–29, Vancouver, Canada, 2005.

E. Charniak. A maximum-entropy-inspired parser. In *Proceedings of the First North American Chapter of the Association for Computational Linguistics*, page 132–139, Seattle, WA, 2000.

N. Chomsky. *Syntactic structures.* Moulton and Co., 1957.

S. Clark and J. R. Curran. Parsing the WSJ using CCG and log-linear models. In *Proceedings of the 42nd Meeting of the Association for Computational Lingusitics*, page 104–111, Barcelona, Spain, 2004.

S. Clark and J. R. Curran. Formalism-independent parser evaluation with CCG and DepBank. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, page 248–255, 2007a.

S. Clark and J. R. Curran. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552, 2007b.

S. Clark, J. Hockenmaier, and M. Steedman. Building deep dependency structures with a wide-coverage CCG parser. In *Proceedings of the 40th Meeting of the Association for Computational Linguistics*, page 327–334, Philadelphia, PA, 2002.

J. M. Cohen. The equivalence of two concepts of categorial grammar. *Information and Control*, 10(5):475–484, 1967. ISSN 0019-9958.

M. Collins. *Head-driven statistical models for natural language parsing.* PhD thesis, University of Pennsylvania, 1999.

C. Culy. The complexity of the vocabulary of Bambara. *Linguistics and Philosophy*, 8 (3):345–351, 1985.

H. B. Curry and R. Feys. *Combinatory Logic, Vol. 1.* North Holland Publishing, Amsterdam, 1958.

V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical logic*, 28(3):181–203, 1989.

P. de Groote. The non-associative Lambek calculus with product in polynomial time. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1617 of *Lecture Notes In Computer Science*, page 128–139, London, UK, 1999. Springer-Verlag Berlin.

M. C. de Marneffe, B. MacCartney, T. Grenager, D. Cer, A. Rafferty, and C. D. Manning. Learning to distinguish valid textual entailments. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*, Venice, Italy, 2006a.

M. C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation*, Genoa, Italy, 2006b.

J. Eisner. Efficient normal-form parsing for Combinatory Categorial Grammar. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, page 79–86, Santa Cruz, CA, 1996.

T.A.D. Fowler. *A graph formalism for proofs in the Lambek calculus with product*. Master's thesis, University of Toronto, 2006.

T.A.D. Fowler. LC graphs for the Lambek calculus with product. In *Proceedings of the 10th Meeting of the Association for Mathematical Linguistics*, UCLA, 2007.

T.A.D. Fowler. Term graphs and the NP-completeness of the product-free Lambek calculus. In *Proceedings of the 14th Conference on Formal Grammar*, Bordeaux, France, 2009.

J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

M. Hepple. Chart parsing Lambek grammars: Modal exstensions and incrementality. In *Proceedings of the 14th conference on Computational linguistics*, page 134–140, Nantes, France, 1992.

J. Hockenmaier. *Data and models for statistical parsing with Combinatory Categorial Grammar*. PhD thesis, University of Edinburgh, 2003.

J. Hockenmaier and M. Steedman. CCGbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33 (3):355–396, 2007.

F. Hoyt and J. Baldridge. A logical basis for the D combinator and normal form in CCG. In *Proceedings of the 46th Meeting of the Association for Computational Linguistics: Human Language Technologies*, page 326–334, Columbus, OH, 2008.

A. K. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of mildly context-sensitive grammar formalisms. *Foundational issues in natural language processing*, page 31–81, 1991.

L. Kallmeyer. Linear context-free rewriting systems. *Language and Linguistics Compass*, 7(1):22–38, 2013.

H. Kamp and U. Reyle. *From Discourse to Logic: An Introduction to Model Theoretic Semantics, Formal Logic and Discourse Representation Theory*. Kluwer Academic Publishers, Dordrecht, Germany, 1993.

D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, page 423–430, 2003.

E. König. The complexity of parsing with extended categorial grammars. In *Proceedings of the 13th conference on Computational linguistics*, page 233–238, Helsinki, Finland, 1990.

M. Kuhlmann and G. Satta. A new parsing algorithm for combinatory categorial grammar. *Transactions of the Association for Computational Linguistics*, 2:405–418, 2014.

M. Kuhlmann, A. Koller, and G. Satta. The importance of rule restrictions in CCG. In *Proceedings of the 48th Meeting of the Association for Computational Linguistics*, page 534–543, Uppsala, Sweden, 2010.

M. Kuhlmann, A. Koller, and G. Satta. Lexicalization and generative power in CCG. *Computational Linguistics*, 41(2):215–247, 2015.

F. Lamarche. Proof nets for intuitionistic linear logic: Essential nets. Technical Report 00347336, INRIA, 2008.

J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958.

J. Lambek. Type grammar revisited. In A. Lecomte, F. Lamarche, and G. Perrier, editors, *Logical Aspects of Computational Linguistics: Selected Papers from the Second International Conference, LACL '97*, Number 1582 in Lecture Notes in Artificial Intelligence, page 1–27. Springer, New York, 1999.

D. Magerman. *Natural language parsing as statistical pattern recognition*. PhD thesis, Stanford University, 1994.

M. P. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330, 1994.

L. T. McCarty. Deep semantic interpretations of legal texts. In *Proceedings of the 11th International Conference on Artificial Intelligence and Law*, page 217–224, Stanford, CA, 2007.

R. McDonald, F. Pereira, K. Ribarov, and J. Hajic. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, page 523–530, Vancouver, Canada, 2005.

R. Montague and R. H. Thomason. *Formal philosophy: selected papers of Richard Montague*. Yale University Press, 1974.

M. Moortgat. Symmetries in natural language syntax and semantics: The Lambek-Grishin calculus. *Logic, Language, Information and Computation*, 4756:264–284, 2007.

R. Moot. Lambek grammars, tree adjoining grammars and hyperedge replacement grammars. In *Proceedings of the 9th International Workshop on Tree Adjoining Grammars and Related Formalisms*, Tübingen, Germany, 2008.

R. Moot. Extraction of type-logical supertags from the spoken dutch corpus. In A. K. Joshi and S. Bangalore, editors, *Supertagging: Using Complex Lexical Descriptions in Natural Language Processing*, page 291–312. MIT Press, 2010a.

R. Moot. Semi-automated extraction of a wide-coverage type-logical grammar for french. In *Proceedings of the Confrence sur le Traitement Automatique des Langues Naturelles*, Montreal, Canada, 2010b.

R. Moot. A type-logical treebank for french. *Journal of Language Modelling*, 3(1): 229–264, 2015.

Glyn Morrill. *Categorial grammar: Logical syntax, semantics and processing*. Oxford University Press, USA, 2010.

G. Penn. A graph-theoretic approach to sequent derivability in the Lambek calculus. *Electronic Notes in Theoretical Computer Science*, 53:274–295, 2004.

M. Pentus. Product-free Lambek calculus and context-free grammars. *The Journal of Symbolic Logic*, 62(2):648–660, 1997.

M. Pentus. Lambek calculus is NP-complete. Technical Report TR2003005, CUNY Graduate Center, New York, 2003.

S. Petrov and D. Klein. Improved inference for unlexicalized parsing. In *Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, page 404–411, Rochester, NY, 2007.

S. Petrov, L. Barrett, R. Thibaux, and D. Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, page 433–440, Syndey, Australia, 2006.

C. Retore. Perfect matchings and series-parallel graphs: Multiplicatives proof nets as R&B-graphs. *Electronic Notes in Theoretical Computer Science*, 3:167–182, 1996.

D. Roorda. *Resource logics: Proof-theoretical investigations*. PhD thesis, University of Amsterdam, 1991.

Y. Savateev. Product-free Lambek calculus is NP-complete. *CUNY Technical Report*, 2008012, September 2008.

S. M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343, 1985.

M. Steedman. Gapping as constituent coordination. *Linguistics and philosophy*, 13(2): 207–263, 1990.

M. Steedman. *The Syntactic Process*. MIT Press, Cambridge, MA, 2000.

L. Tesnière. *Eléments de syntaxe structurale*. Librarie Klincksieck, Paris, 1959.

K. Vijay-Shanker and D. Weir. Polynomial time parsing of combinatory categorial grammars. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, page 1–8, Pittsburgh, PA, 1990.

K. Vijay-Shanker and D. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27(6):511–546, 1994.

W. Xu, S. Clark, and Y. Zhang. Shift-reduce CCG parsing with a dependency model. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, page 218–227, Baltimore, Maryland, 2014.

W. Xu, M. Auli, and S. Clark. 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 (Short Papers)*, page 250–255, Beijing, China, 2015.

W. Zielonka. Axiomatizability of Ajdukiewicz-Lambek calculus by means of cancellation schemes. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 27: 215–224, 1981.