English Coordination in Linear Categorical Grammar

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree
Doctor of Philosophy
in the Graduate School of The Ohio State University

By
Andrew Christopher Worth

/#/

Graduate Program in Linguistics
The Ohio State University
2016

Dissertation Committee:
Professor Carl Pollard, Advisor
Professor Robert Levine, Co-advisor
Professor William Schuler
Coordination is a rich and common phenomenon that occurs when at least two linguistic expressions co-occur with a coordinating conjunction between them. While earlier proposals insisted that the phenomenon was limited to expressions of “like category”, it was later observed in [SGWW85] that the restriction is somewhat weaker, in that the any expression combining with a coordinate structure must be “syntactically construable” with each conjunct.

Logical categorial grammars can broadly be divided into two classes of grammars: Lambek grammars, which make use of the ordered implication connectives \ and /, and currysque grammars, which encode phenogrammar (linear position, roughly) and tectogrammar (syntactic combinatorics) in different components, typically using the typed lambda calculus and its associated intuitionistic implication $\rightarrow$ for the former, and linear implication $\multimap$ for the latter. Lambek categorial grammars have historically provided straightforward analyses of coordination, but have struggled to analyze certain other phenomena such as medial extraction and quantifier “lowering”. By contrast, currysque grammars analyze these phenomena easily, but struggle with directionally sensitive phenomena such as coordination. A number of different proposals have been extended to remedy the deficiencies of Lambek grammars, but the coordination problem has remained intractable for currysque grammars. If the currysque program is to be a viable one for formalizing natural language, this must be addressed.

Linear Categorial Grammar (LCG) is a currysque grammar formalism, and as such, suffers from the concomitant difficulties analyzing coordination. This dissertation discusses the mechanics of LCG, and then extends it to \( \text{LCG}_\varphi \), Linear Categorial Grammar with
Phenogrammatical Subtyping, by adding a restricted form of Lambek and Scott-style subtyping \cite{LS86} to LCG. We specify certain predicates over strings called phenominators, and use these together with subtyping to show how the necessary distinctions can be restored to the phenogrammar in order to enable an analysis of a broad variety of coordinate structures, and we construct a fragment doing exactly this.

Once this has been accomplished, we turn to other, more complex coordination phenomena. First, we discuss so-called predicatives, expressions which can serve as the complements of verbs of predication such as \textit{is} and \textit{became}. We discuss a number of different kinds of predicatives, and show that more is required in order to analyze such constructions. We build on work originating in \cite{Bay96} and continuing with \cite{Mor94} and \cite{PH03}, and add the “additive disjunction” connective $\oplus$ from linear logic to show how this can be used to model the combinatorial properties of verbs of predication. We extend a result from \cite{PH03} by adding the “coordination monad” type constructor $\diamond$, which addresses an inconsistency in the underlying theory, and enables analysis of so-called “unlike category coordination” (UCC). We revise and expand our fragment to cover a large number of examples of UCC. We also provide a brief discussion of argument neutrality, which, while not terribly relevant for English, is nevertheless related to UCC, and we show that a well-known analysis of the phenomenon can be encoded in LCG as well.

We then turn to the understudied area of iterated coordination, or list coordination, wherein a number of nonfinal conjuncts appear in a list, usually with some associated prosody. We show that these constructions can be analyzed by adding the multiset type constructor, which we write $(-)^+$, to the tectogrammar, and the list type constructor, which we write $\,\,\,$, to the phenogrammar along with its associated term constructors cons and the empty list. We revise and expand our fragment to show how examples of iterated coordination can be given an analysis in $\text{LCG}_\varphi$, as well as showing how iterated coordination interacts with unlike category coordination.

Finally, we discuss a number of outstanding issues, and suggest several future avenues.
for research on coordination in LCG. We summarize the results given in the dissertation and conclude that at least with respect to English coordination, the curryesque program is still viable, and that LCG remains a sufficiently expressive grammar formalism.
This dissertation is dedicated to Diana Najjar, who most likely has no idea that by introducing me to Venn diagrams thirty years ago, she inadvertently set me on the course culminating in the present work, and to the memory of Harold Worth, who never knew he helped me get here either.

_He missed the point I had tried to make by writing new language rather than new music for a mass for the dead: In the beginning was the word._

– Kurt Vonnegut
Acknowledgements

My deepest gratitude goes to my advisors Carl Pollard and Bob Levine, who have been with me since almost day one of my career, and without whom it never would have occurred to me to do this in the first place. I have been profoundly changed by the way of looking at the world they have endeavored to instill in me, and touched by their friendship, concern, and camaraderie.

Carl Pollard has served as a better mentor than I could have ever asked for, and possibly better than I deserve. At every step of this journey, he has been there encouraging me to look deeper, think harder, work better, and to clarify and refine my thoughts and insights into their purest form. This is the rarest kind of pedagogy. When I think about this period of my life, the most intellectually delightful times were spent in coffee shops (and more infrequently, dive bars), filling page after page of notebook paper with arcana until neither of us were capable of anything resembling coherent thought. It occurs to me that this is how the pursuit of a doctorate is supposed to feel, and I am forever indebted to him for his efforts to aid me in pursuing the things that I found fascinating. He is careful, funny, patient, supportive, clever, and endlessly kind, and I have kept his terrifying exhortation “Don’t get distracted” in mind daily throughout this process.

My co-advisor Bob Levine was the first to introduce me to the subject matter that would ultimately underlie all of this. I remember fondly sitting in the tiny seminar room and being completely flabbergasted at the fact that one could even begin to tell a story about the deep and profound connections between language and logic. I continue to be grateful for his insight and his periodic appearance in my doorway. “Let me ask you something . . .”
I’m grateful to my current and former committee members David Odden, William Schuler, Shari Speer, Judith Tonhauser, and Mike White for helping me develop my abilities as a writer and independent researcher. Their comments and questions have proved invaluable in enabling me to look more deeply into things I may have otherwise taken for granted. Special thanks are due to Gerald Penn, who helped to formulate the first “real” version of *say*.

A fair amount of this work was completed while I received funding through the Targeted Investment in Excellence (TIE) grant from OSU to our department, and I am pleased and honored to have been the recipient of some of that investment. Kathryn Campbell-Kibler provided me with valuable bonus years, stimulating conversation, and a fascinating project to work on as I completed my dissertation. I offer my best wishes for the continued success of OhioSpeaks / See Your Speech as the years move forward.

I’m grateful to our tireless office staff Brett Gregory, Claudia Morrettini, and Julia Papke, who did their level best to make sure that no requirements slipped through the cracks, and that I always got paid on time. Jim Harmon has been friendlier and more helpful than any IT staffer ever had any right to be. Julie McGory deserves special mention for having been my go-to friend and confidante more times than I can count. Hope Dawson was instrumental in developing my facility as a teacher, and the intellectual and personal rewards from that endeavor defy description.

I owe countless thanks (and even more drinks) to my friends, colleagues, and office-mates Scott Martin and Vedrana Mihaliček, whose thinking has heavily influenced my own, and who have remained steadfast friends. Their companionship and humor have pulled me through at many points when I might have otherwise faltered. I am grateful to Emily Butler, Jon Fintel, Martin Kong, Yusuke Kubota, Matthew Metzger, Andy Plummer, Abby Walker, and Murat Yasavul, all of whom have proved worthwhile and influential conversationalists. Manjuan Duan deserves special mention for making the initial observation that led ultimately to the theory of phenominators.
I used John Dehdari’s “osuthesis” LATEXtemplate to typeset this document, and he deserves praise for sparing me the time and effort of having to figure out how to do it on my own.

My mother, Donna Worth, always believed that I could do whatever I set my mind to, and provided vital and crucial care when it seemed like I might not ever pull this off. She has never wavered in encouraging me to wholeheartedly pursue whatever weird things I find interesting, and her own dogged persistence has served as a model for me on many occasions. She was my first teacher, and I strive to conduct myself in an equally good-willed, sincere, honest, and straightforward manner in all of my dealings with my students, colleagues, and mentors.

My father, Robert Worth, has been my longest and most patient intellectual collaborator. For my entire life he has endeavored to learn something of what I learned, and in turn to share something of whatever he learned. This give-and-take has been integral in developing the sense of fascination and awe that I experience when I think deeply about a problem, and the sense of joy and beauty that I experience when things resolve themselves.

I am grateful to Daurene Worth for her cheerfulness, positivity, and exuberance, which has helped immensely to counterbalance some of my own occasionally maudlin and catastrophizing tendencies. Peter Whitten has been a friend longer than I can remember, and his concern for me and general amazement at what I do have helped me to remember why I ever fell in love with this subject matter in the first place.

When it comes to the love and support of my partner Mollie Workman, I finally run out of words. In the earliest days she was always ready with encouragement and cheerfulness; in the middle days, she shouldered more than her fair share of the burdens of plotting and solidifying the trajectory we share; and in the latter days, she was constantly present to remind me that I could do this, and I would finish, and that yes, I had in fact submitted all paperwork by the relevant deadline. I am especially grateful for the introduction she made between myself and Marcello, my daily collaborator.
None of this would have been possible without all of you.
Vita

M.A. in Linguistics, The Ohio State University, 2012.
Graduate Associate Teaching Award, 2013.
Distinguished University Fellow, The Ohio State University, 2007–2013.
TIE Fellow, The Ohio State University Department of Linguistics, 2010.  
200-level Teaching Award, The Ohio State University Department of Linguistics, 2010.
Graduate Research Associate, The Ohio State University Department of Linguistics, 2013–2015.
Graduate Teaching Associate, The Ohio State University Department of Linguistics, 2008–2012.

Publications


Fields of Study

Major Field: Linguistics
# Table of Contents

Abstract ................................................................. ii
Dedication ............................................................... v
Acknowledgements ...................................................... vi
Vita ............................................................... x
List of Figures .......................................................... xv
List of Tables ......................................................... xvi

1 Introduction ........................................................... 1
   1.1 English coordination ........................................... 1
   1.2 Categorial grammar .............................................. 3
       1.2.1 Lambek grammars ........................................ 4
       1.2.2 Curryesque grammars .................................... 5
   1.3 Fragment methodology ......................................... 7
   1.4 Coordination in Linear Categorial Grammar .................. 8
   1.5 Summary ...................................................... 9

2 Linear Categorial Grammar: Overview ............................. 11
   2.1 Overview .................................................... 11
       2.1.1 Logical ................................................ 12
       2.1.2 Curryesque ........................................... 14
       2.1.3 Sign-based ............................................ 14
       2.1.4 Categorial ............................................ 15
       2.1.5 Parallel-relational ................................... 15
   2.2 Tectogrammar ................................................... 19
       2.2.1 Linear logic ............................................ 19
       2.2.2 Tectotypes ............................................. 20
       2.2.3 On tectoterms ......................................... 21
       2.2.4 Inference Rules ........................................ 22
   2.3 Phenogrammar ..................................................... 23
       2.3.1 Higher order logic ..................................... 24
       2.3.2 Phenotypes ............................................ 26
       2.3.3 Phenoterms ............................................ 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.4 Inference rules</td>
<td>31</td>
</tr>
<tr>
<td>2.4 Semantics</td>
<td>33</td>
</tr>
<tr>
<td>2.4.1 Semantic types</td>
<td>34</td>
</tr>
<tr>
<td>2.4.2 Semantic terms</td>
<td>35</td>
</tr>
<tr>
<td>2.4.3 Inference rules</td>
<td>35</td>
</tr>
<tr>
<td>2.5 Grammar rules</td>
<td>35</td>
</tr>
<tr>
<td>2.6 Examples and sample derivations</td>
<td>37</td>
</tr>
<tr>
<td>2.6.1 Lexical entries</td>
<td>37</td>
</tr>
<tr>
<td>2.7 Discussion</td>
<td>46</td>
</tr>
<tr>
<td>3 Constituent and Nonconstituent Coordination in LCG with Phenominators</td>
<td>47</td>
</tr>
<tr>
<td>3.1 Overview and data</td>
<td>47</td>
</tr>
<tr>
<td>3.1.1 Data</td>
<td>48</td>
</tr>
<tr>
<td>3.1.2 Comparison with previous approaches</td>
<td>50</td>
</tr>
<tr>
<td>3.2 Extending LCG</td>
<td>52</td>
</tr>
<tr>
<td>3.2.1 Phenominators</td>
<td>53</td>
</tr>
<tr>
<td>3.2.2 Fine-grained phenotypes with Lambek and Scott-style subtyping</td>
<td>55</td>
</tr>
<tr>
<td>3.2.3 The Function say and vacuities</td>
<td>59</td>
</tr>
<tr>
<td>3.3 Analysis</td>
<td>66</td>
</tr>
<tr>
<td>3.3.1 Overall strategy</td>
<td>66</td>
</tr>
<tr>
<td>3.3.2 Revised lexical entries</td>
<td>67</td>
</tr>
<tr>
<td>3.3.3 Canonical coordination</td>
<td>70</td>
</tr>
<tr>
<td>3.3.4 Noncanonical coordination</td>
<td>75</td>
</tr>
<tr>
<td>3.4 Discussion</td>
<td>84</td>
</tr>
<tr>
<td>4 Predicatives and Unlike Category Coordination in LCG</td>
<td>85</td>
</tr>
<tr>
<td>4.1 Overview and Data</td>
<td>85</td>
</tr>
<tr>
<td>4.1.1 Predicatives</td>
<td>87</td>
</tr>
<tr>
<td>4.1.2 Coordinate structures composed of predicatives</td>
<td>89</td>
</tr>
<tr>
<td>4.2 Analytical strategy and comparison with previous approaches</td>
<td>90</td>
</tr>
<tr>
<td>4.2.1 Bayer, Johnson, and Morrill</td>
<td>90</td>
</tr>
<tr>
<td>4.2.2 Whitman</td>
<td>91</td>
</tr>
<tr>
<td>4.2.3 Pollard and Hana</td>
<td>92</td>
</tr>
<tr>
<td>4.2.4 LCG</td>
<td>93</td>
</tr>
<tr>
<td>4.3 Technicalia</td>
<td>95</td>
</tr>
<tr>
<td>4.3.1 Categorical monads</td>
<td>96</td>
</tr>
<tr>
<td>4.3.2 Tectogrammar</td>
<td>97</td>
</tr>
<tr>
<td>4.3.3 Phenogrammar</td>
<td>107</td>
</tr>
<tr>
<td>4.3.4 Grammar rules</td>
<td>108</td>
</tr>
<tr>
<td>4.4 Analysis</td>
<td>111</td>
</tr>
<tr>
<td>4.4.1 Lexical entries</td>
<td>111</td>
</tr>
<tr>
<td>4.4.2 Nonlogical rules</td>
<td>114</td>
</tr>
<tr>
<td>4.4.3 Example proof</td>
<td>116</td>
</tr>
<tr>
<td>4.4.4 Argument neutrality</td>
<td>120</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Derivation for Joffrey sniveled ................................. 42
2.2 Derivation for Tyrion slapped Joffrey .......................... 42
2.3 Derivation for Ned gave Arya Nymeria ......................... 43
2.4 Derivation for Olenna poisoned a coward ....................... 44

4.1 Derivation for Robb and Catelyn died .......................... 124
4.2 Derivation for Joffrey whined and sniveled ..................... 125
4.3 Derivation for Joffrey is evil ........................................ 126
4.4 Derivation for Joffrey is Aerys the third ......................... 127
4.5 Derivation for Joffrey is a tyrant ................................. 128
4.6 Derivation for Joffrey is on the iron throne ..................... 129
4.7 Derivation for Joffrey is a coward and a tyrant ................. 130
4.8 Derivation for Joffrey is evil and a tyrant ....................... 131
4.9 Derivation for Joffrey is on the iron throne and a tyrant ........ 132
4.10 Derivation for Joffrey is evil and on the iron throne .......... 133
4.11 Derivation for Joffrey is evil and a tyrant and on the iron throne 134
4.12 Derivation for findet und hilft Frauen ............................ 135

5.1 Derivation for Joffrey became evil, a tyrant, and deceased ........ 153
List of Tables

1.1 Syntactic categories for sentences with NP gaps .......................... 5

2.1 Table of customary (meta)variables ........................................... 18

2.2 Comparison of Linear Logic notation ......................................... 20

3.1 Common phenomenators .......................................................... 54

4.1 Comparison of Linear Logic notation, redux .............................. 98
Chapter 1

Introduction

English coordination is an incredibly common, incredibly rich phenomenon, yet analyzing it has historically proven difficult. One of the noteworthy triumphs of categorial grammar, as exemplified by the tradition originating with [Lam58] and continuing with [Ste85] and [Dow88], has been to provide an account of coordination that is concise, intuitive, and far-reaching in its coverage. However, such grammars have struggled with accounting for many other phenomena, which has given rise to what we refer to as “curryesque” (after [Cur61]) categorial grammars. Linear Categorial Grammar (LCG) is a framework for articulating grammars, or a grammar formalism, which is broadly in this tradition. By contrast, LCG can analyze many kinds of phenomena which have proved to be difficult for other categorial frameworks. Until now, an analysis of anything more complicated than string coordination has not been articulated in any curryesque formalism. We will show that by adding a restricted form of subtyping, LCG will be able to analyze a broad variety of coordinate structures.

1.1 English coordination

The simplest definition of English coordination is that coordination occurs when at least two linguistic expressions co-occur with a coordinating conjunction between them, as in the following examples:

(1) Tyrion and Sansa hated Joffrey.
(2) Joffrey whined and sniveled.
Sansa knew who and what killed Joffrey.

Tyrion slapped and Tywin chastised Joffrey.

Ned gave Bran Summer and Arya Nymeria.

Joffrey is evil and a tyrant.

Joffrey is on the iron throne and a tyrant.

Joffrey is evil and on the iron throne.

Joffrey is evil and a tyrant and on the iron throne.

Joffrey became evil and a tyrant.

Tyrion hated Joffrey, Tywin, and Cersei.

Olenna hated, poisoned, and killed Joffrey.

Joffrey became evil, a tyrant, and deceased.

Coordination is typically observed to obey the principle [PZ86] dubs Wasow’s generalization: “an element in construction with a coordinate constituent must be syntactically construable with each conjunct”. Of course there is much to unpack in this statement, and perhaps one issue to quibble with. First of all, for the purposes of this dissertation, we interpret “coordinate constituent” not as constituency in the traditional phrase-structural sense, but merely in the sense of a particular expression functioning as a grammatical unit. By “grammatical unit” we have a definition in mind which is simultaneously naive and technical; we are referring to the concept of the sign, which is naively modeled as a form-meaning pair, and represented technically for us in a tripartite fashion: first, as the expression’s basic potential to combine with other expressions as observed through co-occurrence; second, the expression’s audible representation and instructions about how to audibly represent it in
combination with other expressions; and third, the compositional meaning of the expression. This definition is made more precise in chapter 2. Finally, we interpret “syntactically construable” simply as “ultimately yielding a grammatical expression under combination”.

1.2 Categorial grammar

We consider here only those grammar formalisms under the rubric of categorial grammar, which is characterized by the assignment of expressions to syntactic categories (types) with a functor / argument structure, and rules specifying how functors and arguments combine. This stands counter to phrase structural traditions, where the locus of combination typically comes from more abstract principles manipulating categories, rather than from the categories themselves. One of the most celebrated historical successes of categorial grammar has been to provide clear, perspicuous, straightforward analyses of coordination phenomena. However, there exist two parallel traditions in categorial grammar, and to our mind, coordination drives a wedge between these traditions.

First, functor coordination phenomena appear to be directionally sensitive, so that points to an analytical and methodological distinction between those grammars which make explicit reference to directionality in their category system (which we generally refer to as Lambek grammars) and those grammars which do not, instead choosing to encode word order in a component distinct from the combinatorial component. We refer to these types of grammars as curriesque, and when the combinatorial component is expressed using linear logic, we refer to them as linear.

The second distinction that coordination impels is between linearity, in the sense of so-called “resource-sensitivity”, and nonlinearity. Linear logic, and grammars based on it have a concrete notion of resource manipulation: one peg goes in one hole, and that’s

---

3One might think of this as something akin to “phonological compositionality” – the audible representation of an expression is a function of the audible representations of the expressions it comprises, and the audible representation of the methods by which it combines with other expressions.
it. For the most part, this conception is useful for constructing grammars for natural language analysis. Coordination, however, seems in general to run counter to this intuition; it enables languages to put more than one expression into a position\(^2\) where one thing would “ordinarily” go. Furthermore, the phenomenon of **iterated coordination**, wherein numerous conjuncts occur in a list is yet another example of nonlinearity. So we are left to conclude that coordination is intrinsically a nonlinear phenomenon, and so grammars must wrestle with this fact.

1.2.1 Lambek grammars

Lambek Grammars, by which we mean those grammars based (minimally) on a rule system containing two varieties of implication, the “leftward-looking” implication \(/\) and the “rightward-looking” implication \(/\), and extensions thereof, have had much success analyzing coordination. In most Lambek Grammars, a given category is interpreted into a set of strings. Larger string sets are built by establishing inference rules operating on categories, and concatenating the strings interpreting those categories. This makes coordination very straightforward, since the essential strategy since [Ste85] is just to take strings which are assigned to like categories and concatenate them with a coordinating conjunction in the middle, yielding a result category of the same category as the two source categories. Additionally, Lambek grammars are highly compositional, and typically have smaller rule sets than phrase structure grammars, since it is the categories themselves together with the inference rules for \(/\) and \(/\) that guide combination.

Owing to the concatenative nature of their phonological representations, Lambek grammars are well-suited to analyzing local phenomena, but become somewhat more cumbersome when addressing nonlocality. In particular, it is not immediately clear how to analyze medial extraction, for example *A prince who Tyrion thinks is a terrible person*. Of course,

\(^2\)We use “position” pre-theoretically here, without meaning anything specific about linear order or tree-based dominance.
numerous augmentations have been proposed to address this issue. Similarly, accounting for quantifier scope ambiguities is less straightforward, since (at least in the simple cases) scope is not determined by string-linear position, so there must be some other method by which quantifiers take scope.

### 1.2.2 Curryesque grammars

The distinction between **tectogrammar**, that is, the abstract grammar of phrasal combination, and **phenogrammar**, or the concrete representation of those combinations, originates in [Cur61]. From the perspective of a theoretician, this is a desirable distinction, because it represents a generalization of the idea of a “syntactic functor” from something which looks left or right to combine with an argument to something which seeks to combine with an argument in a more arbitrary fashion. The following table shows three expressions with “missing” noun phrases together with descriptions of their syntactic categories in Lambek and curryesque systems, respectively. The first two are given different category descriptions in Lambek systems, and the third is (naively) unanalyzable. By contrast, curryesque systems give a uniform description to all three, leaving the spelling out of specific differences to the phenogrammar.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Lambek description</th>
<th>Curryesque description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__ gave Arya the sword</td>
<td>leftward-looking, NP argument</td>
<td>missing NP argument</td>
</tr>
<tr>
<td>Ned gave Arya __</td>
<td>rightward-looking, NP argument</td>
<td>missing NP argument</td>
</tr>
<tr>
<td>Ned gave ___ the sword</td>
<td>?</td>
<td>missing NP argument</td>
</tr>
</tbody>
</table>

Table 1.1: Syntactic categories for sentences with NP gaps

From the perspective of the working linguist, this is a desirable distinction to make, if for no other reason than the fact that there are many languages exhibiting nonconcatenative morphosyntax. Having the flexibility to represent some structure of word / phrase

---
^3A similar idea can also be found in [Dow96].
combinations in multiple ways in principle allows for the articulation of analyses of languages with much freer word order than English, a fact has been explored profitably in [Mih12]. Curry himself observed in [Cur61] that Lambek’s system was restricted only to cases involving concatenation, and suggested that this was an undesirable “admixture of phenogrammatics”.

In contrast to Lambek grammars, linear logic-based curryesque grammars employ only one mode of implication, the unordered linear implication $\rightarrow$. The move to curryesque grammars has been impelled largely by the need to give a more straightforward analysis of phenomena like medial extraction (e.g. *a boy who Cersei wanted ___ to be a king*) and quantification than the ones offered by Lambek grammars. In particular, the tradition beginning with [Oeh94] can be seen as an encoding of the original curryesque program into the linear lambda calculus, as well as an implementation of [Mon73]’s notion of “quantifying in”. In [Mus03], a similar strategy is employed to address medial extraction, among other things, providing a clear account of both kinds of phenomena in a way which falls out automatically from the setup of the theory itself.

This is all well and good, providing there is no reason to make explicit reference to directionality. Unfortunately for the curryesque program, coordination seems to provide exactly such an example. While coordination of simple strings as in (1) is straightforward, things break down rather quickly when it comes to functor coordination. As noted in the table above, a uniform syntactic category is given to all three of the expressions in question. While this is desirable in terms of parsimony, it would naively seem to predict that, tectogrammatically speaking, we ought to be able to construct examples like the following:

(14) *Ned [gave Bran ___] and [___ gave Arya] the sword.

(15) *Ned [gave ___ the sword] and [gave Arya ___].

So these kinds of expressions cannot be distinguished based on their tectogrammar alone.
If we are true to the curryesque program, we should look to the phenogrammar to solve the problem, but the situation there is no better. All three are represented simply as functions over strings, and we have no way of inspecting the functions to check if they have the same “shape”.\footnote{By “shape” we mean, roughly, that, all other things being equal, they have the same number of arguments, and these occur in the same places within the phenogrammatical representation.} So this is not sufficient either. To make matters worse, it does not even appear possible to write down an expression for the phenogrammar of coordination that covers all of the above cases, as well as the many others that occur commonly in English.

So it becomes obvious that the inherently unordered nature of linear implication makes directionally sensitive phenomena like coordination much more difficult, a problem noted in \cite{Kub10} in section 3.2.1., and explored more thoroughly in \cite{KL15} and \cite{Moo15}. So we are faced with a theoretical tension: must we make a choice between the obvious general benefits of curryesque grammars with respect to nonlocal phenomena and the straightforward analysis of coordination provided by Lambek grammars? Can the curryesque program be salvaged? To what extent is it possible to recover some notion of directionality from curryesque grammars?

Assuming that those questions can be answered in the affirmative, in order to evaluate the suitability of the curryesque program for articulating theories of grammar, we should push it to its limits by attempting to account for complicated phenomena to which it seems ill-suited at first glance. To that end, we should have as a primary desideratum a robust theory of coordination, covering as many different kinds of examples as possible. We are constrained to work within what we view as the spirit of the curryesque program: namely, maintaining a strict separation between the tectogrammar and phenogrammar.

\section{Fragment methodology}

In this dissertation, we adopt a fragment methodology, which is a common technique for articulating theories accounting for generalizations of linguistic facts. In essence, we create a
“toy” grammar sufficient to generate expressions exemplifying the data to be accounted for. First, we give the relevant examples, and discuss what we take to be the facts surrounding them. Next we articulate a generalization of the phenomenon in question, and finally we formalize that generalization in our grammatical framework of choice. Second-order questions can then be posed and discussed regarding the suitability of the framework for analyzing linguistic data.

Our fragment takes the form of specifying the nature of the objects we are reasoning about; here, these are linguistic signs, and we formalize them as LCG expressions. We provide both lexical entries for the words occurring in the data to be analyzed, and we provide recursive rules for the combination of signs. Then we are able to give derivations of the examples in question.

Following this, we may bring in subsequent examples to call into question certain choices made along the way, or to add to the complexity and coverage of the fragment. These examples will be accounted for through a combination of extensions to the grammar formalism in the form of the addition of new rules, the addition of new lexical entries, and revisions of the rules and lexical entries given so far, although here we will not engage in any rule revision, and only in minor lexical revision. The full fragment will analyze all of the examples given in section 1.1, and potentially many other examples as well.

1.4 Coordination in Linear Categorial Grammar

In this dissertation we show how to extend Linear Categorial Grammar (LCG), a curryesque grammatical framework, in order to analyze a broad variety of coordinate structures in English. The central technical mechanism underlying this extension is the notion of the phenomenonator, a particular kind of lambda term which will allow us to make our phenogrammatical syntactic categories more fine-grained by defining subtypes distinguishing between expressions which are otherwise category-identical. These subtypes are expressed in the
manner of [LS86], but unlike the general case, for which type-checking is undecidable, our subtypes are constrained in such a way to make decidability of type-checking at least an open question which we discuss in more detail in section 6.1.2.

We further extend the tectogrammatical component of LCG with two operations in order to analyze unlike category coordination (UCC). As is common to many categorial analyses, we extend our grammar with (additive) disjunction types, which are native to linear logic. We show that a standard analysis of unlike category coordination is not quite coherent from a formal standpoint. The types which are generally given to coordinate structures composed of unlike categories are not quite “large enough” to encompass the actual constructions in question. To that end, we describe a primitive strong monad called the coordinate structure monad, and show how it may be used together with disjunction to ramp up types in order to analyze UCC. For the sake of completeness, we show how to extend LCG with additive conjunction types in order to analyze case syncretism in English, and how to use the same technology to encode a well-known analysis of a particular construction from German originating in [Bay96].

Iterated coordination in curryesque grammars is slightly more complex, since the extensions necessary to analyze it take place in more than one component simultaneously. To that end, we show how to extend the tectogrammar with the finite multiset monad, which allows the formation of unordered groups of resources, and how to extend the phenogrammar with the list monad, which allows the formation of ordered groups of resources.

1.5 Summary

Following this introduction, chapter 2 introduces Linear Categorial Grammar with discussions of its representational choices for tectogrammar, phenogrammar, and semantics. We provide grammar rules tying all three of the components together. Subsequently, we give sample derivations for a number of sentences analyzable in the fragment. In chapter 3, we
give an overview for our analysis of coordination. We extend LCG to LCG$\varphi$, LCG with phenogrammatical subtyping. Finally, we show how these extensions can be used to analyze a broad variety of English coordination phenomena. Chapter 4 extends LCG with additive conjunction and disjunction connectives from linear logic, and with a monad type constructor similar to that of [Mog91]. This remedies an architectural problem, and shows how LCG can successfully analyze unlike category coordination. Chapter 5 discusses iterated coordination, and augments the tectogrammar and phenogrammar in a manner sufficient to analyze such constructions. The final chapter summarizes this dissertation, and discusses remaining issues and directions for future research, including a brief exploration of the tantalizing possibility that phenominators can also be used together with list types to analyze intonation in English.
Linear Categorial Grammar: Overview

2.1 Overview

Linear Categorial Grammar (LCG) is a relatively new framework for defining grammars in order to provide formal analyses of natural language phenomena. LCG is a sign-based, logical, parallel-relational, categorial grammatical framework. Abbreviatory overlap has been a constant problem, and LCG is essentially the same as the frameworks referred to as Linear Grammar (LG) and Pheno-Tecto-Differentiated Categorial Grammar (PTDCG), and developed in [Smi10], [Mih12], [Mar13], [PS12], [Pol13], [Wor14], and [PW15]. Lexical entries take the form of the nonlogical axioms of the logical system, and grammar rules correspond to inference rules. LCG is what we refer to as curryesque. In some ways superficially reminiscent of the so-called “T-model” of mainstream generative grammar (MGG), LCG distinguishes between three components of grammatical structure. First, the phenogrammar, which is taken to be a kind of “pre-audible” component; that is, string-based information sufficient to serve as the input to some kind of phonological module. To that end, it is intended to encompass word order, intonation, grammatical stress, audible morphology, and the like. Secondly, LCG has the notion of a tectogrammar, a component which contains information about “traditional” grammatical category: sentence, noun phrase, and so forth, as well as information such as case, number, and person traditionally analyzed in terms of “features”. Furthermore, the tectogrammar specifies the basic combinatorics of the grammar, although each component plays a part in this as well. Finally, LCG has a semantic component intended to model linguistic meaning. LCG belongs to a family
of grammar formalisms known as **categorial grammars**, which broadly encompasses three traditions: Combinatory Categorial Grammar (CCG) [SB11], and the Curryesque and Lambek Categorial Grammar traditions, both of which fall under the rubric of logical categorial grammars. Unlike some of its curryesque kin, in particular [dG01]'s Abstract Categorial Grammar and [Mus10]'s λ Grammar, LCG is **parallel-relational**, meaning that instead of interpreting the terms and types of a tectogrammar into phenogrammar and semantics, respectively, LCG constructs each component simultaneously and independently, though the process is guided and restricted by overarching grammar rules operating on all three components.

### 2.1.1 Logical

LCG is, broadly speaking, a **logical grammar**, meaning that it takes lexical entries to be axioms in some variety or other of logic, and the grammar rules to be rules of inference in said logic. Combination of expressions proceeds by invoking these rules, and the resulting expressions are proved to have particular categories. In this way, expressions correspond to proofs, and inhabited categories correspond to the provable formulas of the logic.

LCG signs are **sequents**, that is, formal statements about the provability of various formulas in a particular logic. As such, we use the “turnstile” $\vdash$ as the primary connective of a sequent $\Gamma \vdash A$, indicating that a particular formula $A$ is provable from the hypotheses $\Gamma$. We refer to $\Gamma$ as the **context**, which is a set of formulas. In LCG, the formulas in question are triples, representing phenogrammatical, tectogrammatical, and semantic information. When the context is empty, we do not typically write anything: a sequent $\vdash A$ should be thought of as $\emptyset \vdash A$. When the context contains a single formula, we may sometimes refer to it as the **antecedent** formula. On right-hand side of the turnstile, the formula which is asserted to be provable is called the **succedent**. In the case of the phenogrammatical and semantic components, these formulas will be annotated with **proof terms**, which can be thought of as recipes for the construction of a proof of the formula in question, or alternately,
as expressions indicating that a particular type is inhabited. These take the form $\Gamma \vdash a : A$, with $a$ a metavariable over terms, and $A$ a type metavariable.

When we are providing LCG signs, which comprise statements about the provability of three separate formulas, we will still use only one turnstile. The technical reader will note that we use $\vdash$ ambiguously, as we systematically suppress the subscripts indicating which signature is relevant for the logic in question (phenogrammatical, tectogrammatical, semantic, and sign-level). Signs are triples, and hypotheses are triples whose terms are variables. A full LCG sign consists of a sequent of the following form:

$$\Gamma \vdash s : S; A; m : M$$

We use the semicolon (;) as the delimiter between grammatical components, and the colon (:) to represent a typing judgment. This sign says that we take $s$ to be a phenoterm of phenotype $S$, with $A$ being the tectotype of the expression, and a semantic term $m$ of semantic type $M$.

In inference rules and grammar rules, we refer to the sequents appearing above the line of inference as the premisses and the sequent below as the conclusion. In the case of elimination rules, the sequent containing the connective which is eliminated from the top of the rule to the bottom is called the major premiss and the other premisses are referred to as the minor premisses of the rule.

We use several pieces of equality-based metalanguage for the sake of brevity. A double colon equals (:=:) is used to define short context-free grammars, generally to say briefly what the language of terms for a given component consists of. The definitional equals $=_{\text{def}}$ allows us to provide general abbreviatory conventions as well as abbreviated representations for more specific, object-language complex terms.
2.1.2 Curryesque

In contradistinction to the so-called “Lambek Categorial Grammars”, which locate the word order primarily in the type system through the directional implication connectives \ and /, we refer to LCG as a Curryesque grammar, after [Cur61], and draw its lineage back to the tradition of categorial grammars originating in [Oeh94]. LCG has as cousins Abstract Categorial Grammar (ACG) [dG01], Lambda Grammar (λG) [Mus10], Hybrid Type-Logical Categorial Grammar [KL12], and others. Curryesque dialects of categorial grammar draw some distinction between Tectogrammar, sometimes called “abstract syntax” (largely in computer science and frameworks heavily influenced by computer science, such as ACG, and the Grammatical Framework of [Ran04]), and Phenogrammar, sometimes called “concrete syntax”.

We conceive of tectogrammar as specifying the most basic combinatorial properties of a grammar, as well as other structural properties such as case, agreement, verb inflectional form, tense, and the like. We locate what is sometimes deemed “phonology” or “prosody” by many categorial theories in the phenogrammar, which broadly encompasses word order, “surface” morphology, intonation, grammaticalized stress, and the like.

2.1.3 Sign-based

LCG is sign-based, in that it takes structures called signs to be the fundamental theoretical objects of linguistic combinatorics. LCG signs consist of triples representing three kinds of information: the tectogrammar, the phenogrammar, and the semantics. While all three components contain logical formulas, the phenogrammar and the semantics additionally contained typed proof terms. This is a slight departure from many other theories of logical categorial grammar, where there is a single type logic, which is decorated with two kinds of terms (what we would deem phenoterms and semantic terms), and its types are interpreted into what we would call phenotypes and semantic types, to which those
terms correspond. In essence, our grammar constructs triples of linguistic information that roughly correspond to sound, combinatoric potential, and meaning.

2.1.4 Categorial

By categorial we generally mean that the expressions of our grammar are assigned syntactic categories qua types in some variety of type theory. In the case of Lambek grammars, these are typically either atomic categories, or categories built with the left and right implication constructors \( \setminus \) and \( / \), to which some add the product (type concatenation or non-commutative fusion) of which these constructors are residuals, sometimes written \( \bullet \), and generally taken to originate with [Lam58]. Traditionally, only tectotypes are called “categories”. Since LCG is tripartite, we assign types in each of the three components. In each case, we build types from a set of linguistically relevant basic types, together with type constructors. In the phenogrammar, these types consist of the type of strings\(^1\), and types of (potentially higher order) functions over strings. In the tectogrammar, they will encompass categories such as NP for noun phrases, and S for sentences, together with a number of connectives mostly drawn from Linear Logic [Gir87]. In the semantics, as should be accessible to the reader familiar with most Montagovian semantic theories [Mon73], the categories will correspond to entities (\( e \)) and propositions (\( p \)), and (potentially higher order) functions over these.

2.1.5 Parallel-relational

In contrast to both labeled deductive systems like those exhibited by grammars in the type-logical tradition such as those described in [Moo10] and [Mor94], and to other Curryesque grammar formalisms, LCG is parallel as opposed to being syntactocentric. Syntactocentric formalisms take syntax as the primary domain within which linguistic combinatorics

\(^1\)These are often left as unanalyzed primitives, and we will not go much beyond that here. The reader should, for the time being, think of these as strings of words. Of course, the picture will become more complicated when one considers intonation and other prosodically-realized linguistic phenomena.
take place, and then sound and meaning are in some way “read off” the syntactic representa-
tion. Instead, we propose three (at least) independent logical systems, one for each
cOMPONENT, whose combinatorics are tied together by overarching grammar rules that spec-
ify the mechanisms by which structure may be built in each component simultaneously.
This means that the character of component interaction is relational, rather than func-
tional, since there is not necessarily a functional mapping between the types of one system
and another.

**Grammar rules**

While each of the logics comprising the type theories of the three components of LCG signs
has its own suite of inference rules, we do not make use of these when reasoning about the
grammar writ large, since LCG signs are composed of information about all three. While it
may be both pedagogically and formally useful to consider each logic’s rules independently,
we will not be able to make any headway as a theory of grammar without rules expressing
how those logics operate together. To that end, we have grammar rules, which come in two
varieties: logical rules and nonlogical rules.

Logical rules are those rules that most closely correspond to the inference rules from
each separate component. They express slight generalizations of those inference rules, and
it is in the nature of these rules that the notion of a grammatical “interface” lives. Logical
rules are those that should be generally familiar to those with a background in categorial
or logical grammar in general, as well as those with some background in computer science.
For the time being, they will consist of three essential rules: the Axiom Schema, which
allows for the creation of hypothetical signs; the rule of Application, which allows for the
combination of signs, and corresponds to implication elimination; and the rule of Abstrac-
tion, corresponding to implication introduction, which introduces hypothetical reasoning
into the system. As we continue it will become necessary to introduce new grammar rules,
but this will be done on an as-needed basis.
Nonlogical rules do not directly correspond to any particular logical rule. Instead, they allow for the establishment of relationships between expressions of certain categories, and other categories. These are akin to what might be deemed “lexical rules” in other frameworks, except in LCG they are not strictly lexical. These could also be thought of in principle as “inaudible” lexical entries. A good example is the mapping of plural nouns to noun phrases, as in [(Dire) wolves] are fiercely loyal or [(Dire) wolves] frolicked in the courtyard.

Conventions

We observe a number of common abbreviatory conventions in order to make things more legible. Except where noted, these conventions will hold for terms and types in general, be they phenogrammatical, tectogrammatical, or semantic.

First and foremost, types are taken to be right-associative, and we will drop parentheses correspondingly. Outermost parentheses are likewise omitted. That is:

\[ NP \to NP \to NP \to S =_{\text{def}} NP \to (NP \to (NP \to S)) =_{\text{def}} (NP \to (NP \to (NP \to S))) \]

Similarly, outermost parentheses in terms are typically omitted, so, for example:

\[ \lambda f. (\lambda x. (f x)) =_{\text{def}} (\lambda f. (\lambda x. (f x))) \]

As was the case with types, abstraction is understood to be right associative, so in conjunction with the previous convention, we have

\[ \lambda f. \lambda x. f \ x =_{\text{def}} \lambda f. (\lambda x. (f \ x)) \]

By contrast, application will be taken to be left-associative as is customary, and we may omit parentheses when no ambiguity results:

\[ f \ a \ b =_{\text{def}} (f \ a) \ b \]
When more than one variable of the same type is bound (by a lambda or quantifier) in sequence, we will use only one binder, and only one judgment about the variable type, in the following way:

\[ \lambda st : St \cdot s \cdot t =_\text{def} \lambda s : St \cdot \lambda t : St \cdot s \cdot t \]

Furthermore, we omit the types of variables when they may be clearly inferred from context, so, for example, in the phenogrammar, one might see

\[ \lambda st : s \cdot t =_\text{def} \lambda st : s \cdot t \]

Finally, we use the notation \( b[a/x] \) to mean “\( b \), with \( a \) replacing every free occurrence of \( x \) in \( b \), changing bound variables as necessary to avoid illicit variable capture”.

The following table gives an indication of what variables we use by convention. When an explicit typing judgment is omitted, the reader may assume that the variable has the type given here.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Use</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, \Delta )</td>
<td>Contexts</td>
<td>Sets or Multisets</td>
</tr>
<tr>
<td>( \tau )</td>
<td>Types</td>
<td></td>
</tr>
<tr>
<td>( \sigma )</td>
<td>Terms</td>
<td></td>
</tr>
<tr>
<td>( S, T )</td>
<td>phenotypes</td>
<td></td>
</tr>
<tr>
<td>( s, t, u )</td>
<td>phenoterms</td>
<td>frequently of type St</td>
</tr>
<tr>
<td>( v )</td>
<td>strings</td>
<td>placeholder for the contiguous string support of a term</td>
</tr>
<tr>
<td>( P, Q )</td>
<td>string functions</td>
<td></td>
</tr>
<tr>
<td>( f, g )</td>
<td>functions</td>
<td>typically pheno-functions</td>
</tr>
<tr>
<td>( \varphi, \psi )</td>
<td>phenominators</td>
<td></td>
</tr>
<tr>
<td>( A, B )</td>
<td>tectotypes</td>
<td></td>
</tr>
<tr>
<td>( M, N )</td>
<td>semantic types</td>
<td></td>
</tr>
<tr>
<td>( m, n )</td>
<td>semantic terms</td>
<td></td>
</tr>
<tr>
<td>( x, y, z )</td>
<td>semantic variables</td>
<td>frequently of type e</td>
</tr>
<tr>
<td>( g )</td>
<td>functions</td>
<td>typically semantic functions</td>
</tr>
</tbody>
</table>

Table 2.1: Table of customary (meta)variables
2.2 Tectogrammar

2.2.1 Linear logic

Linear logic [Gir87] is generally described as being “resource sensitive”, owing to the absence of the structural rules of contraction (which allows for duplication of hypothetical formulas) and weakening (which allows the introduction of specious hypothetical formulas). Often it is referred to as being concerned with resource production and consumption, giving it a character appropriate for modeling natural language, as we can think of certain expressions (verbs, for example) as ‘consuming’ argument expressions (noun phrases) in order to ‘produce’ other expressions (sentences).

Linear Logic is commonly divided along the lines of the so-called “additive” and “multiplicative” connectives. Generally speaking, the multiplicative connectives allow for disjoint contexts, and the additive connectives require like contexts. It may be useful to think of the multiplicative connectives as genuinely linear, and the additives as behaving more like the more familiar versions from intuitionistic logic. Intuitionistic logic (and ultimately, classical logic) is entirely recoverable from linear logic using the “of course” modality, written !, which allows for the controlled reintroduction of contraction and weakening. In fact, varieties of linear logic that split the context into both linear and intuitionistic partitions have been proposed, for example, the systems found in [Wad94], [Ben95], and [BP96].

Another potential source of confusion is due to the fact that there are multiple versions of the LL connectives in use: those originating with [Gir87], and those originating with [Tro92]. In general, distinguishing between the two is straightforward, with the exception of the two disjunctive identities \( \bot \) and 0, which are switched from the Girard connectives to the Troelstra connectives. We will use the Girard connectives here, which are perhaps slightly more common, but we provide the following table so that the reader familiar with only one system may freely translate between the two.
2.2.2 Tectotypes

In LCG, most tectotypes are formulas of implicational linear logic. We will consider certain augmentations in future sections, but for the time being, they should be recognizable to any reader with a passing familiarity with categorial grammar. Instead of using the better known “slashes” of bilinear logic, as is the case with Lambek categorial grammars, LCG uses the “lollipop” (linear implication, symbolized as $\rightsquigarrow$) of linear logic to indicate a non-directional notion of functional dependency. That is, we think of a formula $A \rightsquigarrow B$ as being the kind of expression that is in search of an $A$ in order to construct a $B$, and we make no further demands on it than that (vis à vis linear order).

**Base types**

Initially, for the fragment given here, it suffices to have base types for noun phrases (for which we use NP), common nouns (N), and finite sentences (S). We will freely augment these as necessary as our analysis proceeds. The set of atomic types $\tau_a$ is as follows:
\[ \tau_a ::= \text{NP} \mid \text{N} \mid \text{S} \]

**Type constructors**

We have only one type constructor (for the time being), the linear implication connective \( \rightarrow \). We will often refer to this connective as “functional”, though this is not strictly correct, since linear implication is not modeled as a function. That is, the models we have in mind are category-theoretic, rather than set-theoretic. Nevertheless, we find the metaphor to be useful, and we hope that the reader will allow us the latitude to use such terminology with no further comment. Our inventory of tectotypes is composed of the following, with \( \tau \) a metavariable over types:

\[ \tau ::= \tau_a \mid \tau_1 \rightarrow \tau_2 \text{ (for } \tau_1, \tau_2 \in \tau) \]

Common “functional” tectotypes will be \( \text{NP} \rightarrow \text{S} \), the type of “verb phrases”, or more generally, sentences “missing” a noun phrase in some position; \( \text{NP} \rightarrow \text{NP} \rightarrow \text{S} \), the type of “transitive verbs”, for which the same caveat applies; and \( \text{N} \rightarrow \text{N} \), the type of nominal modifiers. We abbreviate tectotypes in an obvious way, so \( \text{VP} =_{\text{def}} \text{NP} \rightarrow \text{S} \), \( \text{TV} =_{\text{def}} \text{NP} \rightarrow \text{VP} \), \( \text{DV} =_{\text{def}} \text{NP} \rightarrow \text{TV} \), and so on. The reader should note however that because the actual linear order of words is encoded in the phenogrammar, these type abbreviations do not always correspond directly to the expected notion of constituency. For example, both verb phrases and right node raising remnants will be given the tectotype \( \text{VP} \).

**2.2.3 On tectoterms**

The reader familiar with ACG will note the absence of tectoterms, which stands in stark contrast to the phenogrammatical and tectogrammatical components. Since tectoterms serve primarily to encode the gross combinatoric structure of a particular derived expression, they do not clearly represent any particular form of linguistic information. Should this prove
shortsighted\(^2\), it is relatively straightforward to add them back into the system, but for the
time being, it is far simpler to suppress them systematically.

This differs both from Lambek categorial systems such as TLG, which has only one kind
of logical formula, which is annotated with “phonological” or “prosodic” labeling terms on
the one hand, and with semantic labeling terms on the other, and from ACG and \(\lambda\)G, for
which there exists a functional mapping between terms (and types) of the tectogrammatical
logic and both the phenogrammatical and semantic logics. The categorically-inclined reader
will note that since ACG has tectoterms, its mappings from tecto to pheno and tecto to
semantics constitute functors.

### 2.2.4 Inference Rules

The tectogrammatical component of LCG is based around two rules from linear logic:
the rule of linear implication elimination (\(\rightarrow\)-elimination or \(\rightarrow\)E), and the rule of linear
implication introduction (\(\rightarrow\)-introduction or \(\rightarrow\)I), in addition to the axiom scheme that
allows for the creation of hypotheses.

\[
A \vdash A
\]

The axiom scheme from linear logic is quite simple: it says that from a bag of resources
containing only \(A\), we may produce an \(A\). Stated another way, it will follow immediately
from the rule \(\rightarrow\)I, below, that \(\vdash A \rightarrow A\), that is, we can freely produce resources which do
nothing but consume an \(A\) and spit the same \(A\) back out.

\[
\frac{\Gamma \vdash A \rightarrow B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B} \rightarrow \text{E}
\]

The elimination rule resembles the familiar rule of modus ponens or application. It
simply says that if you have a resource \(A \rightarrow B\) (from \(\Gamma\)), that is, one which will produce a \(B\)

\(^2\)For example, it may be useful to consider adding dependent product types to the tectogrammar, for
which tectoterms will be absolutely essential.
when provided with an $A$, then when an $A$ is in fact provided (by virtue of some resources $\Delta$), you will be able to produce a $B$ from $\Gamma$ and $\Delta$ together.

$$\Gamma, A \vdash B \quad \Gamma \vdash A \rightarrow B \rightarrow I$$

The rule of $\rightarrow$-introduction is likewise easy to understand: if you could have produced a resource $B$ by making use of some other resource $A$ (and some bag of resources $\Gamma$), then you can produce (from $\Gamma$ itself) a resource of type $A \rightarrow B$, that is, a resource which will produce a $B$ when provided with an $A$. This rule should be familiar to those with a philosophical or logical background as hypothetical proof, and to those from a computational background as abstraction.

### 2.3 Phenogrammar

For LCG, the phenogrammar of a sign comprises something closer to its audible manifestation. We might consider phenogrammar to be the input to whatever phonological processes will ultimately result in an audible representation. To that end, we specify word order in the phenogrammar, rather than the tectogrammar, as is common in most other Lambek grammars, or by virtue of “spelling out” a series of terminals, as is common in mainstream generative grammar. We leave room for various other prosodic phenomena, including intonation, stress, and sub-word-level morphological phenomena such as cliticization, affixation, and the like. It is our ultimate intention to provide a more complex theory of phenogrammar analyzing many of these phenomena, but such a task is regrettably beyond the current scope.

We choose to axiomatize our theory of phenogrammar in higher order logic, albeit a higher order logic that behaves slightly differently than the reader may be familiar with, due to the nature of its type system and subsequent axiomatization. For the time being, we will locate ourselves at a level where word-strings are taken to be the fundamental
building blocks of a phenogrammatical representation. In addition to strings, we have (potentially higher order) functions over strings. We may sometimes refer to these as the **linearization function** of a particular expression, though we generally reserve that usage only for functions which are lexically specified.

### 2.3.1 Higher order logic

We obtain higher order logic (HOL) from the typed lambda calculus by extending it in a fairly simple way. First, in addition to whatever preexisting base types we had, we add a new type **Bool**, the type of truth values, and then a family of operations \( \tau \vdash \tau \to \tau \to \text{Bool} \), where \( \tau \) (likewise \( \tau_1 \), \( \tau_2 \)...) is a metavariable over types. From these extensions it is possible to define all the familiar quantifiers and connectives as shown in [LS86], which we do here in the manner of [Pol12], using \( p \) and \( q \) as metavariables over HOL formulas, \( x \) as a metavariable over HOL variables, and using \( t \) here only as a variable of type **Bool**:

\[
\text{true} = \text{def} \ast = \ast
\]

This definition relies on the identification of a necessary truth, namely that the single term of the unit type is equal to itself.

\[
\forall x. p = \text{def} \lambda x. p = \lambda x. \text{true}
\]

The truth of a universally quantified statement \( \forall x. p \) is obtained by examining the equality of the function \( \lambda x. p \) with \( \lambda x. \text{true} \). To put it another way, if the function obtained by mapping \( x \) to \( p \) is the same as the function obtained by mapping \( x \) to \( \text{true} \), then we know \( \forall x. p \).

\[
\text{false} = \text{def} \forall t. t
\]

---

\(^3\)As opposed to either the level of sub-word morphology (e.g. clitics, affixes, syllables, etc.), or higher levels of prosodic representation (e.g. minor phrases, intonational phrases, utterances, etc.). It may be useful for the reader inclined to ponder such phenomena to think of our type St as being something more like St(Wd), where Wd is the phenotype of words.
Falsity is defined by any necessarily false formula, in this case $\forall t. \ t$, the formula stating that every boolean $t$ is true.

$$p \land q = \text{def} \langle p, q \rangle = \langle \text{true}, \text{true} \rangle$$

We can define logical conjunction by checking to see if the pair of the two formulas in question is the same as the pair $\langle \text{true}, \text{true} \rangle$.

$$p \rightarrow q = \text{def} \ p = (p \land q)$$

We can observe that the truth conditions of $p = (p \land q)$ mimic precisely those which we would expect for $p \rightarrow q$, and define it thusly.

$$p \leftrightarrow q = \text{def} \ (p \rightarrow q) \land (q \rightarrow p)$$

Unsurprisingly, bi-implication is defined by taking the conjunction of the implications $p \rightarrow q$ and $q \rightarrow p$.

$$\neg p = \text{def} \ p \rightarrow \text{false}$$

The negation of a formula $p$ is defined by the observing the truth of the formula obtained by mapping that formula to false.

$$p \lor q = \text{def} \ \neg (\neg p \land \neg q)$$

Disjunction is defined via De Morgan’s laws.

$$\exists x. \ p = \text{def} \ \forall x. \ \neg p$$

Existential quantification is defined in the same manner.

An important point to be made is that in HOL you can quantify over variables of any type. In chapter 3 we will extend the phenogrammar of LCG to a system called LCG$_\phi$ which is augmented with subtyping in the manner of Lambek and Scott [LS86], where subtypes
are thought of as properties of certain terms within a type, and are defined by giving the characteristic function associated with that property. What remains is to show that the subtype can be embedded within the supertype.

There are a number of ways of axiomatizing HOL to obtain different logical systems, but for our purposes here, it suffices to provide the axioms for lambda conversion:

\[(\alpha\text{-conversion}) \quad \vdash \lambda x: A. b = \lambda y: A. b[y/x] \]

\[(\beta\text{-reduction}) \quad \vdash ((\lambda x: A. b) a) = b[a/x] \]

\[(\eta\text{-reduction}) \quad \vdash \lambda x: A. (f x) = f \text{ (for } x \text{ not free in } f : A \to B) \]

The notation \(b[a/x]\) should be read as “\(b\), substituting \(a\) wherever \(x\) occurs”. By custom when an equation relies on one of these three rules, we will subscript the \(=\) operator with the relevant rule, for example, \(=_{\beta}\).

### 2.3.2 Phenotypes

**Base types**

As mentioned previously, the central notion of our phenogrammatical type system is that of the string. However, we will choose to model these in a manner inspired by the treatment of monoids in category theory, as functions from a unique base type \(m\). LCG has this strategy in common with its cousin ACG [dG01], and it has the pleasing property that strings can be given a normal form. Furthermore, this will allow us to define what will become a central notion in our analysis of coordination: the **phenominator**, a special kind of function over strings, which, due to this definition, will turn out to be simply a particular kind of linear combinator. In addition to the “monoid type” \(m\), we have the unit type 1 and since we are working in higher order logic, we have the type of truth values \(\text{Bool}\).

\[\tau_a := m \mid 1 \mid \text{Bool} \]
Type constructors

We have two type constructors for the time being: the intuitionistic implication →, often pronounced “implies” or “into”, and the product type constructor ×, pronounced “times”.

\[ \tau ::= \tau_a | \tau_1 \rightarrow \tau_2 | \tau_1 \times \tau_2 \text{ (for } \tau_1, \tau_2 \in \tau) \]

Since we will define strings in terms of \( m \) and \( \rightarrow \), we make the following abbreviatory convention:

\[ St =_{\text{def}} m \rightarrow m \]

Furthermore, when we are dealing with \( n \)-ary functions over strings (as opposed to functions over functions), we will adopt the following convention:

\[ St_0 =_{\text{def}} St \]

\[ St_{n+1} =_{\text{def}} St \rightarrow St_n \]

The reader will note that the numeric subscript on the type indicates the number of \( St \)-type arguments the predicate has. This will often, though not always, correspond to the number of \( e \)-type arguments in the semantics. So, for example:

\[ St_2 =_{\text{def}} St \rightarrow St_1 =_{\text{def}} St \rightarrow St \rightarrow St_0 =_{\text{def}} St \rightarrow St \rightarrow St \]

\[ St_1 \rightarrow St =_{\text{def}} (St \rightarrow St_0) \rightarrow St =_{\text{def}} (St \rightarrow St) \rightarrow St \]

2.3.3 Phenoterm

Terms in the phenogrammar consist of atomic terms and complex terms. Atomic terms consist solely of variables and those constants given in the signature, which consists of the basic types and the set of typed nonlogical constants forming the alphabet of the logic. First, we provide a set of distinct typed variables \( v \) :

\[ v =_{\text{def}} v_0, v_1, v_2, \ldots, v_n \]
By custom, we will often use $s, t, u$ and $v$ as metavariables over string variables. Additionally, we will have a set of nonlogical constants $c$, called the **signature** of the logic.\footnote{This quirk of usage is not perfectly precise, as the actual signature is the entire alphabet of symbols over which terms and types are formed.} With respect to the phenogrammar, we generally use this to refer to the set of string constants in particular, for example the strings *TYRION*, *SLAPPED*, and *JOFFREY*. String constants themselves will appear in **SMALL CAPS**, and are always of type $\text{St}$, that is, $m \rightarrow m$. Whenever the reader sees a term written in this manner, it may be safely assumed to be a string, and we will omit a full listing of every constant in the signature, on the grounds that they may be implicitly reconstructed from their use in lexical entries and derivations. We have additionally the sole term of the unit type $\ast$.

Now, with $\sigma$ a metavariable over phenoterms, and $\sigma_1, \sigma_2$ likewise, we can construct complex terms in the following ways:

$$\sigma ::= v \mid c \mid * \mid (\sigma_1 \sigma_2) \mid \lambda v : \tau. \sigma \mid \langle \sigma_1, \sigma_2 \rangle \mid \pi \sigma \mid \pi' \sigma$$

The reader more familiar with the system of higher order logic involving boolean connectives, quantification, and the like will find this system different in character, owing to the fact that the inventory of base types is different. Were we to provide a slightly different axiomatic theory, along with some other base types, defining such connectives would be possible. We will use a system more similar to traditional HOL for the semantic component of our theory in section 2.4. The application term constructor $(\ )$ and the abstraction term constructor $\lambda$ should be familiar, and are associated with the rules of $\rightarrow$-elimination and $\rightarrow$-introduction, respectively. We also provide the pairing type constructor $(\ ,)$ and its associated projections $\pi$ and $\pi'$.

**Strings as monoidal linear terms**

Naively, we could take strings to be primitives, and axiomatize their behavior as a monoid in the usual fashion. We give an associative binary operation on strings $\cdot$, called **concatenate**.
nation and a distinguished element $\epsilon$, called the **empty string**. The empty string $\epsilon$ serves as a two-sided identity for $\cdot$, so that concatenating any string $s$ with the empty string (on either side) results simply in returning $s$. String concatenation and identity are common to many dialects of categorial grammar, among them certain versions of TLG, HTLCG, and ACG. Similarly to ACG, we can define strings in a manner inspired by category theory, as being expressions of type $m \to m$, where $m$ is some distinguished type. Then it becomes poss-
elible to define concatenation as function composition, and the empty string as the identity function. This obviates the need for monoid axioms, as we will show presently.

\[
\begin{align*}
st &= \text{def } m \to m \\
\epsilon &= \text{def } \lambda x : m. x \\
\cdot &= \text{def } \lambda fg : m \to m. \lambda x : m. f (g x) \text{ (written infix)}
\end{align*}
\]

Defined this way, strings are well-behaved, that is, they exhibit the following properties, which indicate that algebraically speaking, strings form a **monoid**.

By way of example, suppose we have strings $s$, $t$, and $u$. Then we can show the following:

**Associativity**

\[
\vdash \forall s : St, t : St, u : St. (s \cdot t) \cdot u = s \cdot (t \cdot u)
\]

remembering that $\text{St} = \text{def } m \to m$, and, for example, $s$ is equivalent to $\lambda x : m. (s x)$, under $\eta$-reduction. Then the first half of the equality is given by following line of reasoning:

\[
\begin{align*}
(s \cdot t) \cdot u &= \text{def } ((\lambda fg : m \to m. \lambda x : m. f (g x) s) t) \cdot u \\
&= _\beta (\lambda g : m \to m. \lambda x : m. s (g x) t) \cdot u \\
&= _\beta (\lambda x : m. s (t x)) \cdot u \\
&= \text{def } ((\lambda fg : m \to m. \lambda x' : m. f (g x') (\lambda x : m. s (t x))) u) \\
&= _\beta ((\lambda g : m \to m. \lambda x' : m. \lambda x : m. s (t x) (g x')) u) \\
&= _\beta (\lambda x' : m. \lambda x : m. s (t x) (u x')) \\
&= _\beta \lambda x' : m. s (t (u x'))
\end{align*}
\]
The second half of the equality is shown by:

\[
\begin{align*}
  s \cdot (t \cdot u) &=_{\text{def}} s \cdot ((\lambda f g : m \to m. \lambda x : m. f (g \, x) \, t) \, u) \\
  &=_{\beta} s \cdot ((\lambda g : m \to m. \lambda x : m. t \, (g \, x)) \, u) \\
  &=_{\beta} s \cdot (\lambda x : m. t \, (u \, x)) \\
  &=_{\text{def}} ((\lambda f g : m \to m. \lambda x' : m. f (g \, x') \, s) \, (\lambda x : m. t \, (u \, x))) \\
  &=_{\beta} ((\lambda g : m \to m. \lambda x' : m. s \, (g \, x')) \, (\lambda x : m. t \, (u \, x))) \\
  &=_{\beta} (\lambda x' : m. s \, ((\lambda x : m. t \, (u \, x)) \, x')) \\
  &=_{\beta} (\lambda x' : m. s \, (\lambda x : m. t \, (u \, x'))) \\
  &=_{\beta} (\lambda x' : m. s \, (\lambda s. s \, (t \, (u \, x'))) \\
  &=_{\eta} t
\end{align*}
\]

Since \(\lambda x' : m. s \, (t \, (u \, x')) = \lambda x' : m. s \, (t \, (u \, x'))\), then \((s \cdot t) \cdot u = s \cdot (t \cdot u)\). Since \(s\), \(t\), and \(u\) were arbitrary, we are justified in concluding \(\forall s, t, u : St. (s \cdot t) \cdot u = s \cdot (t \cdot u)\).}

**Identity**

What remains to show that strings form a monoid is to show that \(\epsilon\) is a two-sided identity for strings, with respect to concatenation. Since concatenation is composition, and \(\epsilon\) is the identity function on strings, this follows trivially, though we provide the proof here for the sake of completeness.

Suppose \(t : St\), then we want to show that \(\forall t : St. \epsilon \cdot t = t = t \cdot \epsilon\). The first equality is given by the following:

\[
\begin{align*}
  \epsilon \cdot t &=_{\text{def}} ((\lambda f g : m \to m. \lambda x : m. f (g \, x) \, \epsilon) \, t) \\
  &=_{\beta} (\lambda g : m \to m. \lambda x : m. \epsilon \, (g \, x) \, t) \\
  &=_{\beta} (\lambda x : m. \epsilon \, (t \, x)) \\
  &=_{\text{def}} (\lambda x : m. \lambda s. s \, (t \, x)) \\
  &=_{\beta} (\lambda x : m. (t \, x)) \\
  &=_{\eta} t
\end{align*}
\]

The second equality is shown in the following line of reasoning:
\[ t \cdot \epsilon =_{\text{def}} ((\lambda f g : m \to m. \lambda x : m. f (g x) \epsilon) \epsilon) \\
=_{\beta} (\lambda g : m \to m. \lambda x : m. t (g x) \epsilon) \\
=_{\beta} \lambda x : m. t (\epsilon x) \\
=_{\text{def}} \lambda x : m. t (\lambda s. s x) \\
=_{\beta} \lambda x : m. (t x) \\
=_{\eta} t \]

Since \( t = t \), and \( t \) was arbitrary, we can conclude that \( \vdash \forall t : St. \epsilon \cdot t = t = t \cdot \epsilon \). So this definition of strings implies that strings form a monoid, as intended.

### String supports

After [Oeh95], when a given term is impure, that is, when it contains string constants (other than \( \epsilon \)), we say that it has a **string support**, or simply **support**. When the support of a (normalized) term consists of a single connected subterm, that is, when all of the string constants happen to be concatenated into a single string, we say that the term has a **contiguous string support**. In virtually every case we are interested in, we will make reference to the contiguous string support, and so we will frequently conflate the two concepts (that is, we refer to a contiguous string support simply as a string support). This idea will turn out to be an important one, as being able to obtain this subterm from a given term will play a large role in the basis of our analysis of coordination. To be precise, for string constants \( s_1, \ldots, s_n \) \((n \geq 0)\), we call \( s_1 \cdot \ldots \cdot s_n \) the **string support** of a term \( \sigma : A \) if \( \vdash \sigma = (\tau s_1 \cdot \ldots \cdot s_n) \) where \( \tau \) is a linear combinator of type \( St \to A \).

### 2.3.4 Inference rules

As was the case with the tectogrammar, there are three inference rules governing the behavior of the phenogrammar in LCG, and they are straightforward analogues of the rules from the tectogrammar. Since our tectogrammar does not have terms, we need to provide some
notion of what to do with proof terms when the inference rules apply, but these too should
be familiar, as function application and abstraction, in addition to the axiom schema.

\[ s : S \vdash s : S \text{ (Ax)} \]

The rule of axiom says, simply, if we assume that we know a proof \( s \) of some formula \( S \),
then from that, we can conclude that \( s \) is a proof of \( S \). This can be seen as a sort of
proof-theoretic version of reflexivity: from a list of proofs consisting of only one proof \( s \), we
are justified in concluding \( s \).

\[ \frac{\Gamma \vdash f : S \rightarrow T \quad \Delta \vdash s : S}{\Gamma, \Delta \vdash (f \ s) : T} \rightarrow E \]

The rule of \( \rightarrow \)-elimination is perhaps the most familiar of all: if we can construct a proof \( f \) of some formula \( S \rightarrow T \), and we can construct a proof \( s \) of \( S \), then by applying \( f \) to \( s \), we are justified in concluding \( T \). Application is the primary method by which expressions combine in LCG, and this rule should be thought of as the subpart which references only
the phenogrammar. Of course, we are generally unable to apply single-component rules,
since LCG signs consist of information in all three components of the grammar.\(^5\)

\[ \frac{\Gamma, s : S \vdash t : T}{\Gamma \vdash \lambda s : T, t : S \rightarrow T} \rightarrow I \]

The rule of \( \rightarrow \)-introduction is only slightly less familiar, allowing for the formation of
functional lambda terms by withdrawing and binding a hypothesis from the context. The
rule says, in essence, that if we assumed a proof \( s \) of the formula \( S \) in order to provide a
proof \( t \) of the formula \( T \), then we can just as well construct a proof of \( S \rightarrow T \) by binding
the hypothesis \( s \) in \( t \).

The rules for the product type will be of less use to us, linguistically speaking, and are
provided here without further comment for the sake of completeness:

\(^5\)It will become necessary later to provide the grammar with a restricted way in which single-component
changes may be allowed to take place, but for the framework overview given here, such technology is unnec-
essary.
\[
\frac{\Gamma \vdash s : S \times T}{\Gamma \vdash \pi s : S} \quad \times E_1
\]
\[
\frac{\Gamma \vdash s : S \times T}{\Gamma \vdash \pi' s : T} \quad \times E_2
\]
\[
\frac{\Gamma \vdash s : S}{\Delta \vdash t : T} \quad \Delta \vdash (s,t) : S \times T \quad \times I
\]

2.4 Semantics

The semantic component of LCG is roughly based on the **Agnostic Hyperintensional Semantics** (AHS of [PP12] and [Pol13]), and it is to these works that we refer the technically inclined reader. Like the phenogrammatical component, it is articulated in higher order logic, albeit with a different type inventory and some nonlogical axioms (meaning postulates). The semantics we give here are impoverished by virtue of the fact that our intent is mainly to provide typed semantic constants as stand-ins for actual, more explicit semantic terms. That is, our semantic terms will show applicative and abstraction relationships, but we do not generally make things more explicit than that. The reason for this is twofold. First and foremost, we wish ultimately to make clear and discuss certain facts about the phenogrammar and the tectogrammar (the conjunction of which comprises what might typically be thought of as “syntax”). Rather than spend a large amount of time on the semantics of coordination, which is an active area of research for categorial grammar, in particular Hybrid TLCG [KL12], we aim to give an account of the phenogrammar and tectogrammar of coordination integrating insights from previous analyses, and improving both coverage and rigor.

The typed constants we provide should be suitable for the working semanticist to articulate analyses by providing meaning postulates. Our aim has been to make the semantics as straightforward and typical as possible, while maintaining an agnosticism about the precise nature of the basis of LCG semantics. The system given here is static, but it should be reasonably straightforward to extend it to a dynamicized theory of the reader’s choice,
such as the one given in [Mar13] and [MP14]. From time to time we will provide small sets of meaning postulates for expository purposes, so that the interested reader may see how LCG semantic terms can be made to resemble more traditional semantic representations. In general, the semantic component as given here should illustrate suppositions about type, scopal relationships, and the like.

2.4.1 Semantic types

While the full type inventory of AHS is richer than that given here, we provide the types necessary for our analysis to be comprehensible. Our type inventory is a subset of the types of AHS, limited to base types \( \tau_a \) for entities and propositions, as well as functional types constructed over those types. The base types of concern are as follows:

\[
\tau_a ::= e \mid p \mid 1 \mid \text{Bool}
\]

We may refer to \( e \)-types as both “individuals” and “entities”, and no theoretical import is to be placed on this distinction for the time being. As was the case with the phenogrammar, our full type inventory is given by virtue of the following typing rule for functional types and product types:

\[
\tau ::= \tau_a \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \quad (\text{for } \tau_1, \tau_2 \in \tau)
\]

We will often abbreviate \( n \)-ary properties, that is, functions from \( n \) entities to propositions, in the following way:

\[
\begin{align*}
 p_0 &= \text{def } P \\
 p_{n+1} &= \text{def } e \rightarrow p_n
\end{align*}
\]

So, for example:

\[
\begin{align*}
 p_2 &= \text{def } e \rightarrow p_1 = \text{def } e \rightarrow e \rightarrow p_0 = \text{def } e \rightarrow e \rightarrow p \\
 p_1 \rightarrow p &= \text{def } (e \rightarrow p_0) \rightarrow p = \text{def } (e \rightarrow p) \rightarrow p
\end{align*}
\]

The reader will note that the numeric subscript conveniently corresponds to the number of \( e \)-type arguments the property has.
2.4.2 Semantic terms

Our semantic terms consist of a countably infinite set of variables $v$, notated $v_0 \ldots v_n$, although we frequently use $x$, $y$, and $z$ by custom. To that we add a set of semantic constants $c$, the sole term of the unity type $\ast$, and then application and abstraction term constructors, followed by pairing and projection constructors. These are defined in the following way, with $\sigma$ a metavariable over semantic terms, and $\sigma_1, \sigma_2$ likewise:

$$
\sigma ::= v \mid c \mid * \mid (\sigma_1 \sigma_2) \mid \lambda \sigma_1 : \tau. \sigma_2 \mid \langle \sigma_1, \sigma_2 \rangle \mid \pi \sigma \mid \pi' \sigma
$$

Semantic constants appear in a sans-serif font, making it easier to discriminate between constants and variables.

2.4.3 Inference rules

The inference rules underlying the semantic component of LCG are in essence identical to those of the phenogrammar, albeit with the metavariables ranging over variables, terms, and types changed to reflect that we are talking about semantics, and so are repeated here without further comment, and with the rules for products omitted.

$$
x : M \vdash x : M \ (Ax)
$$

$$
\frac{\Gamma \vdash g : M \rightarrow N \quad \Delta \vdash m : M}{\Gamma, \Delta \vdash (g m) : N} \rightarrow E
$$

$$
\frac{\Gamma, x : M \vdash n : N}{\Gamma \vdash \lambda x : M. n : M \rightarrow N} \rightarrow I
$$

2.5 Grammar rules

LCG Grammar rules take a similar form to the inference rules from each individual component, except that they express how to manipulate LCG signs in each component simultaneously. Since these are not specifically tied to any particular logical connective, they are
a kind of metalogical inference rule. As was previously noted, we are unable to make use of the inference rules inside each component, since LCG signs are triples. Instead, we need rules that operate point-wise on each element of the LCG sign. The reader will note that the logical inference rules within each component are very similar; this makes it straightforward to generalize to the grammar rules of LCG proper. There are three rules in the basic architecture of the framework: the axiom schema (Ax), the rule of application (App), and the rule of abstraction (Abs).

\[ s : S; A; x : M \vdash s : S; A; x : M \]

The axiom schema allows for the creation of hypothetical LCG signs. The choice of types may be completely independent; that is, a functional type in one component does not necessarily mandate a functional type in another component. This is a departure from many other categorial grammars, both Curryesque and otherwise. The rule states that we can produce a sign comprised of an \( S \)-type phenoterm \( s \), a tectotype \( A \), and an \( M \)-type semantic term, as long we register its assumption in the context.

\[
\Gamma \vdash f : S \rightarrow T; A \leadsto B; g : M \rightarrow N \quad \Delta \vdash s : S; A; m : M \quad \text{App} \\
\Gamma, \Delta \vdash (f s) : T; B; (g m) : N
\]

Upon inspection of each component individually, it becomes clear that the rule of application simply takes the functor term in each component, and applies it to the corresponding argument term. If any component is not a functor, then application may not proceed. When this situation arises, we refer to this as phenogrammatical, tectogrammatical, or semantic blocking, that is, blocking the successful derivation of the sign in question.

\[
\Gamma, x : S; A; y : M \vdash t : T; B; n : N \\
\Gamma \vdash \lambda x : S. t : S \rightarrow T; A \leadsto B; \lambda y : M. n : M \rightarrow N \quad \text{Abs}
\]

Similarly, the rule of abstraction proceeds point-wise in each component, over the occurrence of a hypothetical sign in the context. The implications created are those of each of the component logics, with linear implication in the tectogrammar, and the implication of higher order logic in the phenogrammar.
2.6 Examples and sample derivations

2.6.1 Lexical entries

Common nouns

(16)  
a. \text{\textendash} COWARD : St; N; coward : p_1
b. \text{\textendash} TYRANT : St; N; tyrant : p_1

The lexical signs for the common nouns \textit{coward} and \textit{tyrant} do are typically similar. They are represented simply as strings in the phenogrammar, given the base tectotype N, and their respective semantic constants, which are given the (unary) property type \textit{p}_1.

Names / “simple” noun phrases

(17)  
a. \text{\textendash} AERYS \cdot THE \cdot THIRD : St; NP; aerys-3 : e
b. \text{\textendash} ARYA : St; NP; arya : e
c. \text{\textendash} BRAN : St; NP; bran : e

with the names \textit{Catelyn, Cersei, Joffrey, Ned, Nymeria, Olenna, Robb, Sansa, Summer, Tyrion}, and \textit{Tywin} likewise

We treat names, or “simple” noun phrases as phenogramatically similar to common nouns: that is, both are just strings. Names bear the base tectotype NP, and for our purposes this will suffice, although if we were to introduce pronouns, the picture would become somewhat more complicated. Many dialects of categorial grammar have provided analyses based either on the “additive conjunction” connective \& (or \& in linear logic), or on some method of introducing features. In Curriesque grammars, features are sometimes encoded either by emulation via subtyping, or with a combination of record types and dependent types, as in [dGM07] and [Mih12]. We remain agnostic about which strategy we wish to pursue for the time being, since other facts may later motivate the introduction
of one or the other for independent reasons. Semantically speaking, names are typed as entities, or individuals.\(^6\)

We give one example of an idiomatic name, that is, one whose meaning is conventionalized and noncompositional:

\[(18) \quad \lambda \text{the} \cdot \text{iron} \cdot \text{throne} : \text{St}; \text{NP}; \text{iron-throne} : e\]

The noun phrase the iron throne is treated in precisely the same manner as other proper names: as a string in the phenogrammar, a tectogrammatical noun phrase, and an entity in the semantics. This differs from our standard treatment of quantified noun phrases, which we generally do take to be compositional and to which we will turn shortly.

**Attributive adjectives**

\[(19) \quad \lambda s : \text{St}. \text{evil} \cdot s : \text{St} \to \text{St}; \text{N} \to \text{N}; \text{evil} : p_1 \to p_1\]

Adjectives provide a simple example of a functional type. Tectogrammatically, they map common nouns to common nouns (type \(\text{N} \to \text{N}\)). In the phenogrammar, they are functions from strings to strings. Upon inspection of the phenoterm, it is easy to see that they simply linearize their argument to the immediate right of their string support, in this case the string constant evil. Since we do not give meaning postulates in this work, we instead provide the semantic constant evil, a function from properties to properties.

**Prepositions**

\[(20) \quad \lambda t s : \text{St}. s \cdot \text{on} \cdot t : \text{St} \to \text{St} \to \text{St}; \text{NP} \to \text{N} \to \text{N}; \text{on} : e \to p_1 \to p_1\]

Tectogrammatically speaking, we make the simplifying assumption that prepositions take noun phrase arguments in order to produce common noun modifiers of type \(\text{N} \to \text{N}\), the same type as adjectives, although many other possibilities exist. Likewise, once they

\(^6\)This suggests a direct reference theory of names. For a more nuanced discussion, the reader should consult [Pol13], [PP12], or [Pol08].
receive their first semantic argument, an entity, they produce functions from properties to properties. In the phenogrammar, they are binary functions on strings that place their first argument to the immediate right of their string support on, and their second argument to its immediate left. At this point the forward-thinking reader may rightly ask how adjectives and prepositional phrases differ in their type. At the moment, they do not, although it should be evident upon inspection that they behave differently with respect to their linearization structure, since in adjectives, their contiguous string support appears to the left of their arguments, and in prepositional phrases, to the right. It will become necessary to make this generalization more precise in the near future. For the time being, it suffices to note that in LCG, the difference in “directionality” lives essentially at the level of terms, rather than at the level of types, as is more common to the Lambek-inspired frameworks.

Verbs

\[\begin{align*}
(21) & \quad a. \quad \lambda s : St. \cdot \text{died} : St_1; VP; \text{die} : p_1 \\
& \quad b. \quad \lambda s : St. \cdot \text{whined} : St_1; VP; \text{whine} : p_1 \\
& \quad c. \quad \lambda s : St. \cdot \text{sniveled} : St_1; VP; \text{snivel} : p_1
\end{align*}\]

The three intransitive verbs above are straightforward. In the phenogrammar, each is a function from strings to strings, linearizing their sole argument to the left of their string support. Tectogrammatically speaking, they are of type VP, meaning that they take arguments of type NP and produce resources of type S. In the semantics, they are simply unary properties of individuals, as expected.

\[\begin{align*}
(22) & \quad a. \quad \lambda st : St. \cdot \text{chastised} \cdot s : St_2; TV; \text{chastise} : p_2 \\
& \quad b. \quad \lambda st : St. \cdot \text{hated} \cdot s : St_2; TV; \text{hate} : p_2 \\
& \quad c. \quad \lambda st : St. \cdot \text{killed} \cdot s : St_2; TV; \text{kill} : p_2 \\
& \quad d. \quad \lambda st : St. \cdot \text{slapped} \cdot s : St_2; TV; \text{slap} : p_2 \\
& \quad e. \quad \lambda st : St. \cdot \text{poisoned} \cdot s : St_2; TV; \text{poison} : p_2
\end{align*}\]
Phenogrammatically, transitive verbs linearize their first string argument to their immediate right, and their second to their immediate left, resulting in a string. The tectotype indicates that the verb expects two NP-type arguments in order to yield a sentence (type S). Unsurprisingly, the verb is a binary property in the semantics. The reader will note that after the transitive verb as taken its first argument, it is structurally similar to an intransitive verb, or verb phrase, as is to be expected. It would be possible of course to analyze transitive verbs as taking their subjects arguments first, and then their object arguments. The choice to take objects first is merely one which is in line with mainstream practice and we do not wish to take any particular theoretical position on this issue.

\[
\lambda stu : \text{St}, u \cdot \text{gave} \cdot s \cdot t : \text{St}_3; \text{DV}; \text{give} : p_3
\]

Like transitive verbs, the phenos for ditransitive verbs specify that string arguments occur both to the left and right of the verb string itself. In this case, the first argument is linearized to the immediate right of the verb string, and the second to the right of that. Practically speaking, this makes the linearization function, once it has been applied to the first argument, structurally similar to that of a transitive verb, and to that of a verb phrase after it has been applied to the second. In the tectogrammar, the verb takes three noun phrase (type NP) arguments in order to produce an S-type sentence. Finally, in the semantics, the relation that the verb specifies is the 3-ary \text{give} relation, a property of three \text{e}-type individuals. As noted previously, there are other theoretically possible orders to linearize a ditransitive, and our choice here is without theoretical weight.

Determiners

\[
\lambda s : \text{St}, \lambda P : \text{St} \to \text{St}, P \cdot (\lambda s) : \text{St} \to \text{St}_1 \to \text{St}; \\
N \to \text{VP} \to \text{S}; \\
a : p_1 \to p_1 \to p
\]
The typing for determiners is somewhat complex, as determiners create quantified noun phrases when they combine with common nouns, and quantified noun phrases are given continuized types, that is, types of expressions which “take scope” within their continuations, which should be thought of as a way of reifying the functional context within which they may eventually occur. This analysis is in essence the same as the one found in [Oeh94], albeit expressed in a slightly different grammar formalism. This process should be familiar to semantically inclined readers, and those familiar with categorial grammar in general, where it is often better known as type-raising. In examining the phenotype of the determiner, we can see that it takes first a string argument, which it linearizes to its immediate right. This argument corresponds to a common noun in the tectogrammar, and a property in the semantics. Once the determiner has combined with its first argument, the resulting expression is a quantified noun phrase, which we examine more detail here, though strictly speaking, they represent not lexical entries, but theorems.

**Quantified noun phrases**

(25)  

a. \( \lambda P. P (a \cdot \text{coward}) : \text{St}_1 \rightarrow \text{St}; \text{VP} \rightarrow S; \text{a coward} : p_1 \rightarrow p \)

b. \( \lambda P. P (a \cdot \text{tyrant}) : \text{St}_1 \rightarrow \text{St}; \text{VP} \rightarrow S; \text{a tyrant} : p_1 \rightarrow p \)

The quantified noun phrases that result from a determiner applying to its common noun are continuized types: they are functions looking for a functional argument, which they then subsequently apply within their own body somewhere. This is true of all three components of the grammar. In the phenogrammar, we see that the term \( a \cdot \text{Coward} \) has a functional argument \( P \) applied to it, with \( P \) lambda-bound. In the tectogrammar, we see that we have an expression seeking a VP-type argument (type NP \( \rightarrow S \)) in order to produce a sentence (type S). Similarly, in the semantics, the quantified noun phrase takes a property argument, which it will subsequently predicate of some part of the subterm of the meaning postulate associated with the semantic constant in question.
In the following proofs we make full use of the abbreviatory conventions discussed in section 2.1.5.

(26) Joffrey sniveled

\[
\begin{array}{c}
\vdash \lambda s : \text{St.} \cdot s \cdot \text{SNIVELED} : \text{St}_1; \text{VP}; \text{snivel} : p_1 \\
\vdash \text{JOFFREY} : \text{St}; \text{NP}; \text{joffrey} : e \\
\vdash \text{JOFFREY} \cdot \text{SNIVELED} : \text{St}; \text{S}; \text{snivel joffrey} : p
\end{array}
\]

Figure 2.1: Derivation for *Joffrey sniveled*

This is about the simplest derivation possible in LCG.\(^7\) We can observe that LCG makes no distinctions between intransitive verbs and VPs, except to note that intransitive verbs have lexical entries, and the category of VPs may include more complex derived expressions. We see the verb phrase *sniveled* combining with its subject argument *Joffrey*. In the phenogrammar, the string *Joffrey* is positioned to the left of the string *sniveled* due to the linear position of the bound variable *s*, resulting in the string *JOFFREY-SNIVELED*. In the tectogrammar, we can see that we produce a sentence (type S) when the VP (type NP → S) applies to its NP-type argument. In the semantics, we note that the unary property *snivel* is predicated of the individual *joffrey*, producing a proposition.

(27) Tyrion slapped Joffrey

\[
\begin{array}{c}
\vdash \lambda st : \text{St.} \cdot t \cdot \text{SLAPPED} \cdot s : \text{St}_2 \\
; \text{TV} \\
; \text{slap} : p_2 \\
\vdash \text{JOFFREY} : \text{St}; \text{NP}; \text{joffrey} : e \\
\vdash \text{TYRION} \cdot \text{SLAPPED} \cdot \text{JOFFREY} : \text{St}_1 \\
; \text{VP} \\
; \text{slap joffrey} : p_1 \\
\vdash \text{TYRION} \cdot \text{SLAPPED} \cdot \text{JOFFREY} : \text{St} \\
; \text{S} \\
; \text{slap joffrey tyrion} : p
\end{array}
\]

Figure 2.2: Derivation for *Tyrion slapped Joffrey*

\(^7\)Aside from those consisting of a single axiom as both leaf and root
Only slightly more complicated is an example involving a transitive verb. Here, we first apply the verb *slapped* to its object argument *Joffrey*. In the phenogrammar, this has the effect of placing the string *JOFFREY* to the right of the string corresponding to the verb, creating the contiguous string support *SLAPPED·JOFFREY*. The remaining phenogrammatical string argument is bound in a position to the left of the support, leaving us with a functional phenoterm identical to the original one in the previous example (26), differing only in the string support. In the tectogrammar, the verb consumes an NP-type argument, to yield an expression looking to combine with another NP in order to produce a sentence (type S). Semantically, the verb is a binary property of individuals, and we specify one of those individuals here, namely, the semantic constant *joffrey*.

Next, things proceed much in the same way as the previous example. We apply the VP expression to its subject argument *Tyrion*. This results in the positioning of the string *TYRION* to the left of the string corresponding to the VP, and ultimately resulting in the single string *TYRION·SLAPPED·JOFFREY*. Tectogrammatically, the NP-type resource is used to construct a sentence. Finally, in the semantics, we can see that the *slap* property is asserted to hold of the individuals *tyrion* and *joffrey*, thus creating a proposition.

(28) Ned gave Arya Nymeria

An example featuring a ditransitive verb provides no particularly new challenges. We apply the verb *gave* first to its indirect object *arya*. Phenogrammatically, the string *ARYA*
is linearized to the immediate right of the string GAVE, creating the support GAVE · ARYA. The remaining two string variables occur to the immediate right and left, in that order, meaning that this expression now has the same phenogrammatical “shape” as a transitive verb. In the tectogrammar, the first NP argument is taken, creating an expression looking for two remaining NPs in order to produce an expression of tectotype S. In the semantics, everything is as expected: the constant give is a 3-ary property of individuals, of which arya is the first.

From this point on, everything proceeds analogously to the two previous derivations in (26) and (27). The direct object nymeria is picked up next, with the subject ned being the final argument. This process ultimately results in the completed derived sign, which reports that phenogrammatically, NED · GAVE · ARYA · NYMERIA is a string, that the overall expression is a sentence of tectotype S, and that semantically speaking, we have the proposition give arya nymeria ned.

(29) Olenna poisoned a coward

Here we see our first example of a higher order function, in the scoping expression a coward. The insight that Curryesque grammars provide a natural mechanism by which to
explain quantifier scope ambiguity lies originally with [Oeh94], to the best of our knowledge, and it has been explored further in [Mus03] and [MP12]. Essentially, the strategy is that quantified noun phrases like *someone* and *everyone* are lexically type-raised in each component of the grammar. Now, owing to the fact that our framework has the rules of Axiom, and Abstraction, we can freely create expressions with “gaps” in virtually any position. Since the difference in linear order between a subject gap and an object gap exists only in the phenoterm, which will correctly place its string arguments in the desired position regardless, we freely allow there to be no difference in tectogrammatical type. So QNPs, which are part of a class of scoping expressions, are “lowered” into position by virtue of the fact that they predicate functional arguments of subparts of themselves. The strategy of allowing nonlogical axioms to stand in for traces is vaguely reminiscent of the G/HPSG tradition, as in [PS94], and the LCG turnstile is analogous to the G/HPSG slash feature, although slash was used exclusively for syntactic dependencies.

The proof begins with an instance of the rule of Axiom, instantiating a hypothetical sign constructed of the pheno variable $u$, of type St, the tectotype NP, and the semantic variable $x$ of type e. That is, this hypothesis corresponds to what we expect a noun phrase to look like. We apply the transitive verb *poison* to its object argument, our hypothetical sign, placing $u$ in the relevant phenogrammatical position to the right of the verb’s string support, and creating a VP sign, albeit one with a hypothetical argument still in the context. Next we apply the VP to its subject argument *Olenna*, which, as before, creates a sentence represented by the phenogrammatical string $\text{olen} \cdot \text{poisoned} \cdot u$, the tectotype S, and the proposition *poison* $x$ *olen*. The reader should note that both the phenogrammatical and semantic terms contain the relevant variables from the hypothetical sign, $u$, and $x$ respectively. Next, we abstract over the hypothetical sign, constructing a function from strings to strings in the phenogrammar which is tectogrammatically a sentence missing a noun phrase (type NP $\rightarrow$ S), and a unary property of individuals. This step is analogous to the trace-binding aspect of movement (both overt and covert) in mainstream generative
grammar (MGG).

Now, we apply the determiner a to its noun argument coward, creating a quantified noun phrase. In examining the type of this QNP, we can see that in the phenogrammar, it takes as its sole argument a unary string function P, which it will apply to the string a·COWARD, ultimately returning a string. Effectively, this allows the string a·COWARD to appear wherever P has its string variable bound. Tectogrammatically, it is looking for a sentence with an NP gap in it, in order to produce a sentence (since intuitively, we will be plugging the gap with the QNP). In the semantics, the QNP will take scope over its unary property argument, creating a proposition. So as we provide the higher order QNP with its functional argument, we apply the phenogrammatical function to the string support of the QNP, creating the string OLENA·POISONED·A·COWARD, a sentence in the tectogrammar, and the proposition (a coward) (λx: e. poison x olena). This entire process can be thought of as a lambda calculus implementation of Montague’s [Mon73] notion of “quantifying-in”.

2.7 Discussion

In this chapter we have introduced Linear Categorial Grammar, a parallel-relational, sign-based, Curryesque, logical grammar formalism. We have introduced the three main components: phenogrammar, tectogrammar, and semantics. We have discussed how to construct strings in the phenogrammar, and we have refined the notion of string supports [Oeh95] in a manner preliminary for extending the formalism in subsequent chapters. We have given examples of lexical entries and derivations illustrating some of the main features of the framework, including the ease with which quantified noun phrases may scope.
Chapter 3

Constituent and Nonconstituent Coordination in LCG with Phenominators

3.1 Overview and data

Linear Categorial Grammar (LCG) is a sign-based, curryesque, relational, logical categorial grammar whose central architecture is based on linear logic. Curryesque grammars separate the abstract combinatorics (tectogrammar) of linguistic expressions from their concrete, audible representations (phenogrammar). LCG has a combinatorial component based on a fragment of linear logic. Most curryesque grammars encode linear order in string-based lambda terms, in which there is no obvious way to distinguish “right” from “left”. Without some notion of directionality, grammars are unable to differentiate, say, subject and object for purposes of building functorial coordinate structures.

Linear logic-based systems have been called “resource-sensitive” due to the fact that they require all hypotheses to be used once and only once throughout the course of their derivations. This makes them particularly suitable for providing analyses of systems where information comes in discrete chunks, and thus particularly suitable for analyzing natural language. However, owing to nondirectional nature of this very resource sensitivity, coordination has been very little studied in the context of such grammar formalisms, as it seemingly flies in the face of the notion. In Lambek CGs, flexibility to the notion of constituency (vis à vis which expressions “count” as belonging to syntactic categories) in conjunction with introduction (and composition) rules have enabled such grammars to
successfully address an entire host of coordination phenomena in a transparent and compositional manner. While “Curryesque” CGs as a rule do not suffer from some of the other difficulties that plague Lambek CGs, many are notably deficient in one area: coordination. Lest we throw the baby out with the bathwater, this is an issue that needs to be addressed.

In this section, we provide an overview and discuss the data to be examined. We compare our approach to previous approaches. Section 3.2 discusses proposed extensions to LCG in order to obtain LCG_\(\varphi\), LCG with fine-grained phenotypes. In section 3.3, we articulate our analysis of coordination in LCG_\(\varphi\) and provide derivations for the examples given. Section 3.4 summarizes and discusses remaining issues for future work.

3.1.1 Data

We take the following to be an exemplary subset of the relevant data, and adopt a fragment methodology to show how they may be analyzed. In this section, we repeat example numbering from chapter 1.

(1) Tyrion and Sansa hated Joffrey.

(2) Joffrey whined and sniveled.

(3) Sansa knew who and what killed Joffrey.

We first examine so-called “constituent coordination”, although this distinction is somewhat less meaningful in a categorial setting than in a phrase-structural setting, as we will quickly see. In (1), which we refer to generally as ‘NP-coordination’, the noun phrases Tyrion and Sansa are coordinated. We may refer to this as ‘string coordination’ when we mean the phenogrammatical representation of the construction, although that description is more general, as many other kinds of expressions are strings, phenogrammatically. The next example (2) showcases VP-coordination, where the intransitive verb (phrases) whined and sniveled are coordinated. The reader will note that these are both single-word VPs; this is theoretically insignificant, as our analysis will generalize to multiword VPs as long
as they have a contiguous string support, a technical notion defined in section 3.2. This example is our first example of functor coordination: coordination of expressions whose phenos are functional, rather than simply being strings. Finally, example (3) shows coordination of wh-words used to form embedded questions. While this is another example of functor coordination, we will see that the structures being coordinated are somewhat more complicated. Nevertheless, our analysis will account for this example in a straightforward way.

(4) Tyrion slapped and Tywin chastised Joffrey.

(5) Ned gave Bran Summer and Arya Nymeria.

We now turn to the apparently more complicated examples referred to generally as “nonconstituent coordination” (although some authors, e.g. [Dow88], use “nonconstituent conjunction” to refer to the phenomenon we call argument cluster coordination here). In (4), we have so-called right node raising, henceforth RNR, which looks something like VP-coordination in reverse. That is, each conjunct is a transitive verb together with its subject, with the object being the material shared between them by virtue of the coordination. This kind of coordination, while mystifying from the approach of a MGG-like framework with a reified notion of constituency, is well known to be fairly uncomplicated for categorial frameworks to analyze, and LCG will be no exception. The final example we consider here is shown in (5), a phenomenon known under several names which we refer to by the generally agreed upon term argument cluster coordination (ACC). In these kinds of examples, two proximal arguments of a particular verb are conjoined as though they were a single unit. Categorial grammars, having a flexible notion of consistency, have had much success providing analyses for coordinations of this form, although they become somewhat more complex than the other kinds of coordination discussed here. It is unsurprising, then, that this example turns out to be the most difficult to account for in LCG as well, and requires more complicated formal techniques in order for us to be able to analyze it successfully.
3.1.2 Comparison with previous approaches

Categorial grammars with directionality

Grammars which feature types that make explicit reference to directionality typically do not have much trouble analyzing coordination. Steedman famously laid the groundwork for such analyses in [Ste85], using Combinatory Categorial Grammar (CCG), and his analysis was revised and extended successfully in [Dow88]. Rather than using full introduction rules for the directional implication connectives \( / \) and \( \searrow \), CCG uses a combination of type-raising and function composition in order to analyze coordination, both of which end up being provable in most frameworks that fully embrace hypothetical reasoning in conjunction with directional implication.

Equivalent and extended analyses of coordination can be found in the tradition of Lambek grammars with hypothetical reasoning, augmented with other connectives, which we refer to as extended Lambek grammars. Chief among these are the logical categorial grammars as exemplified by Morrill’s Type Logical Grammar (TLG) and his Displacement Calculus, formulated in [Mor94] and in [MVF11] respectively, and explored in [Whi02]. Another line of research in this tradition is provided by the Hybrid Type Logical Categorial Grammar (HTLCG, here, HTLG) of Kubota and / or Levine, as discussed in [KL15b], [KL12], [KL13], and [Kub15]. HTLG is ‘hybrid’ in two ways: it mixes both (bilinear) directional and linear (non-directional) modes of implication, and it has some separation between tectogrammar and phenogrammar, though this difference only comes to light with respect to linear implication, setting it at least partially at odds with curryesque grammars, at least in the strict sense of [Cur61].

All of these frameworks locate at least some of their notion of directionality in their type logics, in the component we would refer to as the tectogrammar. This is perhaps slightly a misnomer, since with the exception of HTLG, these grammars do not generally make a distinction between phenogrammar and tectogrammar, although the “structures” formed
as the antecedents of sequents do some duty as a phenogrammatical representation. It is
our intention to explore the curryesque program in as pure a form as possible, maintaining
a strict separation between the combinatorics of expressions, and the potentially audible
representations they may ultimately attain. To that end, we restrict the implication of
the tectogrammar to linear implication, and make every attempt to locate the actual di-
rectionality of the expressions in the phenogrammar. Certainly directional CGs have had
commendable success analyzing coordination, but they tend to have more trouble with
quantification, medial extraction, and other kinds of constructions involving non-peripheral
expressions.

Curryesque grammars

For curryesque grammars, the situation is exactly the opposite. They have no trouble
analyzing constructions with medial extraction and quantification, for example, but coor-
dination provides a problem with no immediately obvious solution. The central reason for
this is that most Curryesque CGs encode linear order in lambda terms, and there is no
obvious way to distinguish ‘right’ from ‘left’ by examining the types (be they linear or in-
tuitionistic).¹ This is not a problem when we are coordinating strings directly, as [dGM07]
show, but an analysis of the more difficult case of functor coordination remains elusive.²

In principle curryesque grammars both overgenerate and undergenerate. Without some
notion of directionality, grammars are unable to distinguish between, say, subject and
object. This would seem to predict, for example, that \( \lambda s. s \cdot \text{SLAPPED} \cdot \text{JOFFREY} \) and
\( \lambda s. \text{TYRION} \cdot \text{SLAPPED} \cdot s \) would have the same syntactic category (NP \( \rightarrow \) S in the tectogram-
mar, and St \( \rightarrow \) St in the phenogrammar), and would thus be compatible under coordination,
but this is generally not the case. Furthermore, it is unclear even how to write a lambda

¹A noteworthy exception is Ranta’s Grammatical Framework (GF), explored in, e.g. [Ran04] and [Ran09].
GF also makes distinctions between tectogrammar and phenogrammar, though it has a somewhat different
conception of each.
²A problem explicitly recognized by [Kub10] in section 3.2.1., and explored thoroughly in [Moo15].
term that capture our intuitions about how coordination should work in anything approaching a general manner. What we need is a way to examine the structure of a lambda term independently of the specific string constants that comprise it. To put it another way, in order to coordinate functors, we need to be able to distinguish between what [Oeh95] calls their string support, that is, the string constants which make up the body of a particular functional term, and the linearization structure such functors impose on their arguments.

One solution has been proposed in [Kan15] for Abstract Categorial grammar (ACG). This paper proposes adding features to ACG that encode ‘leftness’ and ‘rightness’, and then using said features to emulate Lambek grammars in ACG. This too is a deviation from the curried program, since it in effect builds directionality back into the tectogrammar. Furthermore, since this strategy is essentially an encoding of the Lambek slashes into ACG, it is still not clear how to extend this analysis to cases of coordinated wh-words or quantifiers, for example.

### 3.2 Extending LCG

Since it is not immediately possible in LCG to differentiate by type alone two functions that are ‘looking for their argument’ in different positions, as it is in (extended) Lambek grammars, we must develop some way to describe this difference. To that end, we aim to encode this difference in functional subtypes of phenogrammatical expressions. So we will need to add a subtyping mechanism, which we will draw in essence from the kind of type theory proposed in [LS86]. Once we have subtypes, we will be able to describe certain subtypes that capture the distinctions we have in mind. These will make reference to objects we call phenominators, which are certain kinds of string predicates that we will use to distinguish between terms that are otherwise of like type. Next, we will add inference rules for manipulating subtypes: an upcasting rule for moving from a subtype to a supertype, and a downcasting rule to move from a supertype to a subtype.
This will get us a good measure of the way there. However, in order to write the lexical entry we have in mind for coordination, we will need to add one further piece of technology: the ability to extract the string support from a given function. We will recursively define a function called *say* that does precisely this (when possible). We will find it necessarily to recursively define inter alia certain additional terms called *vacuities* that will be used in the process of reducing terms to their support. Finally, having made these definitions, we will be able to write a lexical entry for *and* that captures the relevant distinctions and enables LCGϕ to analyze coordination of both strings and functors, restoring what was lost in the move from Lambek grammars to curryesque grammars.

3.2.1 Phenominators

The center of our analysis of coordination is the notion of a *phenominator* (short for *pheno-combinator*), a particular variety of typed lambda term. Intuitively, phenominators serve the same purpose for LCG that bilinear (slash) types do for Lambek categorial grammars. Specifically, they encode the linearization structure of a functor, that is, where arguments may eventually occur with respect to its string support. To put it another way, a phenominator describes the structure a functor “projects”, in terms of linear order. We will call a phenoterm a *combinator* if it is a closed lambda term containing no logical constants.\(^3\) We refer to a term as *linear* if each variable occurring within it is bound exactly once. We would like to define an A-phenominator as a closed linear lambda term of type St → A (for A a metavariable over phenotypes), i.e. a term whose first argument is a string, and containing no nonlogical constants other than concatenation and the empty string.

Due to the string encoding given in chapter 2, this means that the phenominators are just\(^3\)

\(^3\)Note that this in principle allows for the presence of logical constants, which may differ from the standard conception of “combinators”. In particular, we will need to make use of the term constructor ↑ϕ, denoting a canonical injection from subtypes to supertypes, which is defined in terms of ∃ and ¬, themselves defined in terms of * and =.
a subset of the linear combinators (augmented with $\uparrow_\varphi$). The idea is that phenominators in some ways describe the abstract “shape” of possible string functions.

For those accustomed to thinking of “syntax” as being word order, then phenominators can be thought of as a kind of syntactic combinator. We use $\varphi$ used by custom as a metavariable over phenominators. For each phenominator $\varphi$ and string $s$, we refer to $s$ as the string support (or sometimes just the support) of $(\varphi \ s)$, and of any term provably equal to $(\varphi \ s)$.

We will generally abbreviate phenominators by the construction with which they are most commonly associated: $n$ for common nouns and noun phrases, $i$ for verb phrases and intransitive verbs, $t$ for transitive verbs, $d$ for ditransitive verbs, $q$ for quantified noun phrases, $w$ for wh-words forming embedded questions, and $r$ for right node raising constructions. Here are examples of some of the most common phenominators we will make use of and the abbreviations we customarily use for them:

<table>
<thead>
<tr>
<th>Phenominator</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda s.s$</td>
<td>$n$</td>
</tr>
<tr>
<td>$\lambda v.s \cdot v$</td>
<td>$i$</td>
</tr>
<tr>
<td>$\lambda v.s.t \cdot v \cdot s$</td>
<td>$t$</td>
</tr>
<tr>
<td>$\lambda v.s.t.u \cdot v \cdot s \cdot t$</td>
<td>$d$</td>
</tr>
<tr>
<td>$\lambda v.P.(P \ v)$</td>
<td>$q$</td>
</tr>
<tr>
<td>$\lambda v.P.v \cdot (P \ e)$</td>
<td>$w$</td>
</tr>
<tr>
<td>$\lambda v.s \cdot v$</td>
<td>$r$</td>
</tr>
<tr>
<td>$\lambda v.s.P.(P \ v \cdot s)$</td>
<td>$\text{det}$</td>
</tr>
<tr>
<td>$\lambda v.P.s \cdot v \cdot (P \ e)$</td>
<td>$\text{rel}$</td>
</tr>
</tbody>
</table>

Table 3.1: Common phenominators

As indicated previously, the first argument of a phenominator will always correspond to what we refer to (after [Oeh95]) as the string support of a particular term. That is, for any term with a string support, it is possible to calculate the relevant phenominator by replacing that support with a string variable, and binding that variable. With the first argument dispensed with, we have chosen the argument order of the phenominators out
of general concern for what we perceive to be fairly uncontroversial categorial analyses of English grammatical phenomena. That is, transitive verbs take their object arguments first, and then their subject arguments, ditransitives take their first and second object arguments, followed by their subject argument, etc. We do not place any theoretical weight on this notational choice, and as long as the arguments in question are immediately adjacent to the string support at each successive application, it is possible to permute them to some extent without losing the general thrust of the analysis. For example, the choice to have transitive verbs take their object arguments first is insignificant.

Since strings are implicitly under the image of the identity phenominator \( \lambda s.s \), we will consistently omit this subscript.

As a consequence of these abbreviations, we are able to give a pleasingly legible representation for the phenos of many LCG / LCG_\( \varphi \) signs by writing a given phenominator next to its string support, e.g. (i whined), (t slapped), etc.

Once we extend LCG to LCG_\( \varphi \), our strategy for analyzing coordination is fairly simple. First, we modify the lexical entries for coordinating conjunctions in order to ensure that the terms being coordinated are under the image of the same phenominator. Then, we need to define a function called `say` that extracts the string support of each term. Next, we concatenate the two supports, with the conjunction between them. Then, we re-apply the phenominator under which the original terms occurred, in order to construct the coordinated term. Finally, we “downcast” back into the relevant subtype, verifying that the resulting term has the same “shape” as each of the conjuncts.

### 3.2.2 Fine-grained phenotypes with Lambek and Scott-style subtyping

We refer to types which exhibit the phenominator subtyping as **fine-grained phenotypes**, and we refer to types which do not have phenominators as **coarse-grained phenotypes**. Likewise, when we speak of fine-grained terms, we mean terms whose types are fine-grained.

\(^4\)Since we believe it is possible to embed Lambek categorial grammars in LCG, this fact reflects that the calculus we are dealing with is similar to the **associative** Lambek Calculus.
In order to augment our type theory with the relevant subtypes, we turn first to [LS86], who hold that one way to do subtyping is by defining predicates that amount to the characteristic function of the particular subtype in question, and then ensuring that these predicates meet certain axioms embedding the subtype into the supertype. We will be able to write such predicates using phenominators. Since we have defined phenominators in such a way to limit them to those whose first argument is a string (i.e., the argument corresponding to the string support of the term occurring under the image of the phenominator), with this idea in place, we are able to assign subtypes to functional types in the following way.

In addition to the inventory of type constructors already described previously, we add the following: if \( A \) is a type and \( P \) is an \( A \)-predicate (i.e. a term of type \( A \to \text{Bool} \)), then \([x : A \mid P \; x]\) is a type, called the subtype of \( A \) determined by \( P \). The set-theoretically inclined reader should understand this as determining the subset of a particular set relative to its characteristic function. The embedding of a subset into its superset is represented in the type theory by the kernel term constructor \( \ker_P : [x : A \mid P \; x] \to A \). Since we have a very restricted collection of types in mind, this is somewhat more general than is necessary, so we adopt a number of conventions for clarity and legibility.

As previously noted, an \( A \)-phenominator is a linear combinator of type \( \text{St} \to A \). Now, for a given \( A \)-phenominator \( \varphi \), we denote by \( P_\varphi \) the \( A \)-predicate \( \lambda f : A. \exists s : \text{St}. f = (\varphi \; s) \), which constitutes a subtyping predicate in the manner of [LS86]. We define fine-grained types \( A_\varphi \) in the following way:

\[
A_\varphi = \text{def } [f : A \mid P_\varphi \; f]
\]

Furthermore, instead of \( \ker_P \), we write \( \uparrow_\varphi \) (read ‘up sub \( \varphi \)’). Additionally, if \( A_\varphi \) is a fine-grained phenotype and \( a : A \) a term such that \( \vdash_\star P_\varphi \; a \) (where \( \vdash_\star \) means provability in the higher order pheno theory, not provability in the grammar), then we write \( \downarrow_\varphi \; a \) (read ‘down sub \( \varphi \) of \( a \)’) for the unique element of \( A_\varphi \) whose image in \( A \) is \( a \). As a consequence of these definitions, the following hold:
1. ⊢₁ ∀p : Aϕ. (↓ϕ (↑ϕ p)) = p

2. ⊢₂ ∀p : A. (Pϕ p) → (↑ϕ (↓ϕ p)) = p

Let us consider the following (putative) lexical entry (pheno only):

\[ \vdash λs'. s' \cdot \text{SNIVELED} : \text{St}_1 \]

We would like to show then that \( P_i \) holds of the term in question, and we are justified in annotating it with \( ↓_i \) and providing it with the type \((\text{St}_1)\), along the following lines:

\[
Aϕ := (\text{St}_1),
\]

\[
:: := (\text{St}_1)_{λs.s'}
\]

\[
P_i := λf : \text{St}_1. \exists t : \text{St}. f = (λs.s \cdot v t)
\]

So applying the subtyping predicate to the term in question, we have

\[
(λf : \text{St} \to \text{St}. \exists t : \text{St}.
\]

\[
f = (λs.s \cdot v t) (λs'. s' \cdot \text{SNIVELED})
\]

\[
= \exists t : \text{St}. λs'. s' \cdot \text{SNIVELED} = (λs.s \cdot v t)
\]

\[
= \exists t : \text{St}. λs'. s' \cdot \text{SNIVELED} = λs. s \cdot t
\]

\[
= \exists t : \text{St}. λs. s \cdot \text{SNIVELED} = λs. s \cdot t
\]

which is true with \( t = \text{SNIVELED} \), and so \((↓_i λs'. s' \cdot \text{SNIVELED}) : (\text{St}_1)\) is shown to be well-typed.

Now that we are dealing not just with coarse-grained phenotypes, but with fine-grained subtypes, the actions of the grammar rules become less clear. In the case of application, since our functional types are now subtypes of functional types, it becomes necessary to say how they may apply to their arguments, and how to calculate (fine-grained) return types. Similarly, with abstraction, we need to be able to discern what fine-grained type, if any, is borne by a term in the conclusion of the rule. In practice, these will amount to giving rules of upcasting and downcasting, allowing us to move between supertypes and subtypes in a safe manner. We will show that while upcasting is straightforward, downcasting is potentially unsafe, and we provide a strategy for introducing it in a principled way.
Embedding and upcasting

It should be obvious that there is a canonical embedding $\ker_{\varphi} : A_{\varphi} \hookrightarrow A$ of a fine-grained type (subtype) into a coarse-grained type (supertype) that maps each term to its own image. Since we make use of Lambek and Scott-style subtyping rather than stipulating subtyping relationships via an ordering relation $\leq$ on types, a standard subsumption rule is unnecessary. We can make do instead with the following derived rule, where we use $\uparrow_{\varphi}$ as a more convenient abbreviation for $\ker_{\varphi}$:

$$
\frac{\Gamma \vdash x : A_{\varphi}}{\Gamma \vdash (\uparrow_{\varphi} x) : A} \quad \text{Embedding}
$$

Since our grammar rules do not make explicit reference to subtypes, we need a rule corresponding to the phenogrammatical embedding rule:

$$
\frac{\Gamma \vdash s : S_{\varphi}; A; m : M}{\Gamma \vdash (\uparrow_{\varphi} s) : S; A; m : M} \uparrow
$$

In practice, this rule simply erases the phenominator, embedding the fine-grained phenotype within its coarse-grained supertype. Without such a rule, it is not immediately clear how to calculate the return types when applying terms of finely-grained types to their arguments. Since what we are concerned with are subtypes of functional types, if the return type is itself functional, we generally lose the subtyping information at each application. To put it another way: without making some changes, return types are always coarse-grained. A possible solution is to annotate types with phenominators “all the way down”, specifying the relevant subtype of each successive functional type. This seems cumbersome. Additional problems are presented by the rule of application, which does not construct fine-grained types. An analytical consequence of this fact is that analyses making use of the abstraction rule (such as our version of right node raising, inspired by [Dow88] and adapted from [Car97]) become unavailable. This is a defect to be remedied presently.
**Downcasting**

In principle, what is needed is a safe way to get from the coarse-grained type to one of its fine-grained subtypes, should the term in question satisfy the predicate which is the characteristic function of that subtype. With this technology, the proper role of abstraction is restored. Additionally, the presence of this rule will allow us to give a kind of abbreviated inference rule, showing phenominators reduce in a predictable manner as functions are applied to their arguments.

\[
\Gamma' \vdash x : A \quad \Gamma' \vdash P_\varphi x \\
\frac{}{\Gamma' \vdash (\downarrow_\varphi x) : A_\varphi} \text{Subtyping}
\]

\[
\Gamma \vdash s : S ; A \vdash m : M 
\frac{}{\Gamma \vdash (\downarrow s) : S_\varphi ; A ; m : M} \downarrow
\]

The reader should note in the above rules that \(\Gamma'\) refers to the projection of the context in its phenogrammatical component, and that \(\vdash_s\) denotes provability in the higher order pheno theory, rather than provability within the grammar itself.\(^5\)

### 3.2.3 The Function **say** and vacuities

In order to write the lexical entry for coordination, we need to define a function we call **say** which extracts from a given term its string support, if such a thing is possible. To be precise, we will define a family of functions \(\text{say}_A\) where \(A\) ranges over what we will call the **sayable** phenotypes. In essence, the idea behind **say** is to keep applying a function to vacuous arguments until its string support is all that is left. To that end, we will define certain objects we call **vacuities**, which are themselves defined recursively. The idea of a vacuity is that it be in some way an “empty argument” to which a functional term may apply. Not all types have vacuities; we call those that do **vacuable** types. If we are dealing with functions taking string arguments, it seems obvious that the vacuity on strings should be the empty string \(\epsilon\). If we are dealing with second-order functions taking \(\text{St} \rightarrow \text{St}\)

\(^5\)This raises issues of decidability with respect to proof search which are taken up in section 6.1.2.
arguments, for example, quantified noun phrases like *everyone*, then the vacuity on \( St \rightarrow St \) should be the identity function on strings, \( \lambda s.s \). Higher vacuities than these become more complicated, and defining all of the higher-order vacuities is not entirely straightforward, as certain types are not guaranteed to have a unique vacuity. We conjecture that the term “vacuity” may just be shorthand for “unique linear combinator”, but research into this claim is ongoing. At any rate, it is possible to calculate a vacuity for any (higher-order) function taking as an argument a function under the image of a phenominator. The vacuity on such a function is just the phenominator applied to the empty string.\(^6\) The central idea is easily understood when one asks what, say, a vacuous transitive verb sounds like. The answer seems to be: by itself, nothing, but it imposes a certain order on its arguments. One practical application of this clause is in analyzing so-called “argument cluster coordination”, where this definition will ensure that the argument cluster gets linearized in the correct manner. We will subsequently provide an analysis wherein the notion of the phenominator is profitably employed in order to adopt and reinterpret a categorial account along the lines of the one given in [Car97].

Vacuities of a particular type can be roughly partitioned into two groups: those that are unique, and those that must be determined by a phenominator, which we refer to as fine-grained. The reader will note that this implies that there are types which do not exhibit vacuities. The unique vacuities are those that belong to what we refer to as **left-associative** types, namely types having the following form:

\[ \text{left-associative} \]

\(^6\)It has been suggested to the author by an anonymous reviewer of the related work [Wor14] that this concept may be related to the “context passing representation” of [Hug95], and the association of a \texttt{nil} term with its continuation with respect to contexts is assuredly evocative of the association of the vacuity on a phenominator-indexed type with the continuation of \( \epsilon \) with respect to a phenominator.
\begin{align*}
\text{St} \\
\text{St} \to \text{St} \\
(\text{St} \to \text{St}) \to \text{St} \\
((\text{St} \to \text{St}) \to \text{St}) \to \text{St} \\
\vdots
\end{align*}

These types may be abbreviated in the following way:

\begin{align*}
\text{St}_0 &= \text{def} \text{St} \\
\text{St}_{n-1} &= \text{def} (\text{St}_n) \rightarrow \text{St}, \text{ for } n \leq 0
\end{align*}

For example,

\begin{align*}
\text{St}_{-1} &= (\text{St}_0) \rightarrow \text{St} = (\text{St}) \rightarrow \text{St} = \text{St} \rightarrow \text{St} \\
\text{St}_{-2} &= (\text{St}_{-1}) \rightarrow \text{St} = ((\text{St}_0) \rightarrow \text{St}) \rightarrow \text{St} = ((\text{St}) \rightarrow \text{St}) \rightarrow \text{St} = (\text{St} \rightarrow \text{St}) \rightarrow \text{St} \\
\text{St}_{-3} &= \ldots = ((\text{St} \rightarrow \text{St}) \rightarrow \text{St}) \rightarrow \text{St}
\end{align*}

Intuitively, the absolute value of the negative subscript captures the order of the string function, i.e., \( \text{St}_{-1} \) is a first-order predicate, \( \text{St}_{-2} \) second-order, and the like. We use negative subscripts here to differentiate these types from the right-associative string functions for which we use positive subscripts, as in the type \( \text{St}_3 \), which abbreviates \( \text{St} \rightarrow (\text{St} \rightarrow (\text{St} \rightarrow \text{St})) \). So positive subscripts denote arity with respect to string functions, and negative subscripts denote order. We refer to types \( A \rightarrow A \) as \textit{endotypes} or \textit{endotypical}.

We say a type is \textit{vacuable} iff it is fine-grained, endotypical, or left-associative. Next, we define the set of vacuities \( \text{vac}_A \) in the following way, where \( A \) and \( B \) are types, and \( \varphi \) a phenominator:

\begin{align*}
(30) \quad &\text{For } A = B_\varphi \text{ fine-grained, } \text{vac}_A = \text{def} \downarrow (\varphi \epsilon) \\
(31) \quad &\text{For } A = B \rightarrow B, \text{ vac}_A = \text{def} \lambda f : B. f \\
(32) \quad &\text{For } A \text{ left-associative:} \\
&\text{a. } \text{vac}_{\text{St}_0} = \text{def} \epsilon
\end{align*}
b. \( \text{vac}_{\text{St} \rightarrow \text{St}} = \text{def} \lambda h : \text{St}_{n-1}. h \text{ vac}_{\text{St}_n} \)

The first clause simply says that the vacuity on types whose terms are under the image of a phenominator is just that phenominator applied to the empty string. In practice, this amounts to imagining a term like that of, say, a transitive verb, whose string support is inaudible, i.e., replaced by the empty string. Since every string is implicitly under the identity phenominator \( \lambda s.s \), the reader should note that as a special case of this clause, we have

\[
\text{vac}_{\text{St}} = \text{vac}_{\text{St}_{\lambda \cdot s.s}} = (\lambda s.s \epsilon) = \epsilon
\]

which gives us the 0-order vacuity (or the vacuity on \( \text{St}_0 \)), the empty string. Furthermore, we can show that the first order vacuity of type \( \text{St}_{-1} \) is also unique, since if we were to calculate the vacuity on functions from strings to strings, we note that there are only two possible phenominators for functions of the type in question (\( \text{St} \rightarrow \text{St} \)): the “VP” phenominator \( \lambda v.s \cdot v \) and the “RNR” phenominator \( \lambda v.s \cdot s \). Both can be shown to reduce by this definition to the obvious vacuity on functions of that type, the identity function \( \lambda s.s \):

\[
\begin{align*}
\text{vac}_{(\text{st} \rightarrow \text{st})_{\lambda \cdot v.s \cdot v}} &= (\lambda v.s \cdot v \epsilon) = \lambda s.s \cdot \epsilon = \lambda s.s \\
\text{vac}_{(\text{st} \rightarrow \text{st})_{\lambda \cdot v.s \cdot s}} &= (\lambda v.s \cdot s \epsilon) = \lambda s.s \cdot \epsilon = \lambda s.s
\end{align*}
\]

In fact, this is more generally true of all types of the form \( B \rightarrow B \), since the identity functions are unique for each type \( B \).

We can compute a vacuity for the phenotype of quantified noun phrases, \( (\text{St} \rightarrow \text{St}) \rightarrow \text{St} \), written \( \text{St}_{-2} \):

\[
\begin{align*}
\text{vac}_{\text{St}_{-2}} &= \lambda P : \text{St}_{-1}. (P \text{ vac}_{\text{St}_0}) \\
&= \lambda P : (\text{St}_0) \rightarrow \text{St}. (P \text{ vac}_{\text{St}_0}) \\
&= \lambda P : (\text{St}) \rightarrow \text{St}. (P \text{ vac}_{\text{St}_0}) \\
&= \lambda P : \text{St} \rightarrow \text{St}. (P \text{ vac}_{\text{St}}) \\
&= \lambda P : \text{St} \rightarrow \text{St}. (P \epsilon)
\end{align*}
\]
Likewise, we can easily do the same for the continuation of a QNP, for example type-raised VP of phenotype \((\text{St} \to \text{St}) \to \text{St}\) to \text{St}, written \(\text{St}_{-3}\):

\[
\text{vac}_{\text{St}_{-3}} = \lambda p : \text{St}_{-2}. (P \text{vac}_{\text{St}_{-1}})
\]
\[
= \lambda p : (\text{St}_{-1}) \to \text{St}. (P \text{vac}_{\text{St}_{-1}})
\]
\[
= \lambda p : ((\text{St}_0) \to \text{St}) \to \text{St}. (P \text{vac}_{\text{St}_{-1}})
\]
\[
= \lambda p : ((\text{St}) \to \text{St}) \to \text{St}. (P \text{vac}_{\text{St}_{-1}})
\]
\[
= \lambda p : (\text{St} \to \text{St}) \to \text{St}. (P \lambda s. s)
\]

and so on, for higher types.

There are types for which no vacuities exist, namely those which are not purely left-associative, and whose terms are not under the image of a phenominator, for example the type \(\text{St} \to \text{St} \to \text{St}\), for which the reader will note is not inhabited by a unique linear combinator. There is no straightforward way to define such terms, since we would need a more expressive notion of phenominator, which somehow encodes the position of all arguments within a term, taking us further into the metalanguage.

Then we can now describe the sayable phenotypes:

1. \(\text{St}\) is sayable.

2. Any fine-grained phenotype \(A_\varphi\) is sayable if \(A\) is sayable.

3. If \(A\) is vacuable and \(B\) is sayable, then \(A \to B\) is sayable.

4. Nothing else is sayable.

Now that we finally have vacuities, we are able to define \textit{say} in the manner described earlier (with \(A, B\) types, and \(\varphi\) a phenominator, as before):

\[
(33) \quad \text{a. } \text{say}_{\text{St}} =_{\text{def}} \lambda s. s
\]
\[
\text{b. } \text{say}_{A_\varphi} =_{\text{def}} \lambda p : A_\varphi. \text{say}_A(\uparrow_\varphi p), \text{ for } A_\varphi \text{ fine-grained and } A \text{ sayable.}
\]
c. \( \text{say}_{A \rightarrow B} = \text{def} \ \lambda p : A \rightarrow B. \text{say}_B(p \ \text{vac}_A) \)

The first clause of the definition grounds the recursion, and says that if you want to \text{say} a string, you simply apply the identity on strings to it. The third clause says that, in order to \text{say} a type under the image of a phenominator, you simply drop the phenominator, and apply the version of \text{say} for the supertype. This may seem counterintuitive at first, but the reader should remember that since phenominators are concerned with describing the linear position of arguments, and the entire idea of \text{say} is to recursively generate vacuous arguments to potentially higher order functions over strings, we do not actually care \textbf{where} a vacuous argument gets placed. The second clause expresses the fact that in order to apply \text{say} to a term which has been downcast, i.e. a term whose type is fine-grained, you must first upcast that term and then apply \text{say} to the result.

The third clause is the primary recursion, and is the most complex. In order to \text{say} a function \( k \) of type \( \tau_1 \rightarrow \tau_2 \), you apply that function \( k \) to the vacuity on its first argument \( \tau_1 \), and then apply the \text{say} function for the return type \( \tau_2 \) to the result. Thus the valence is reduced by one argument, and the recursion continues, bottoming out at the type \( St \), the simplest type of our phenogrammar at this point.\(^7\)

It can be shown to be a consequence of the definitions that for any sayable type \( A \) and \( A \)-phenominator \( \varphi \), and for any string \( s \):

\[ \vdash \text{say}_A(\varphi \ s) = s \]

i.e. \( \text{say}_A \) is a left inverse for every \( A \)-phenominator, and that for every \( n \geq 0 \),

\[ \vdash \text{say}_{St^{-n}} = \text{vac}_{St^{n-1}} \]

For an expedient example, we can apply \text{say} to our putative lexical entry from earlier, and verify that it will reduce to the string \text{SNIVELED} as desired:

\(^7\)Technically, \( St \) is not a base type, but is defined as \( m \rightarrow m \) in chapter 2.
say_{St_1} \lambda s. s \cdot SNIVELED
= \lambda p : St_1. (say_{St}(p \ vac_{St})) \lambda s. s \cdot SNIVELED
= say_{St} (\lambda s. s \cdot SNIVELED \ vac_{St})
= say_{St} (\lambda s. s \cdot SNIVELED \epsilon)
= say_{St} \epsilon \cdot SNIVELED
= say_{St} SNIVELED
= \lambda s. s \ SNIVELED
= SNIVELED

The reader may note that there is an asymmetry to the definitions of say on the one hand and vac on the other; that is, it is possible in principle to construct expressions which do not ultimately achieve a specific string representation. For example, our phenogrammar allows the derivation of a term like the following: \lambda f : St_2. ((f JOFFREY) TYRION) : (St_2) \rightarrow St. If we attempt to go through the recursion above by applying say, we can see that we reach a term which cannot be reduced further:

say_{(St_2) \rightarrow St} \lambda f : St_2. ((f JOFFREY) TYRION)
= \lambda p : (St_2) \rightarrow St. say_{St} (p \ vac_{St_2}) \lambda f : St_2. ((f JOFFREY) TYRION)
= say_{St} (\lambda f : St_2. ((f JOFFREY) TYRION) \ vac_{St_2})
= say_{St} (\vac_{St_2} JOFFREY) TYRION)
= \lambda s. s ((\vac_{St_2} JOFFREY) TYRION)
= ((\vac_{St_2} JOFFREY) TYRION)

This is because \vac_{St_2 \rightarrow St} is not subject to the definition of vacuities, by virtue of the fact that it lacks a (unique) phenominator. This is by design; we require vacuities to be associated with a particular phenominator for precisely this reason. Without prior understanding of where to linearize the arguments of a given function, we will of course be unable to say it. The situation is remedied by specifying a relevant phenominator:
\begin{align*}
\text{say}_{(St_2)_t \rightarrow St} \lambda f : (St_2)_t \cdot (((\uparrow_t f) \text{ JOFFREY}) \text{ TYRION}) \\
= \lambda p : (St_2)_t \rightarrow St. \text{say}_{St} (p \text{ vac}_{(St_2)_t}) \lambda f : (St_2)_t \cdot (((\uparrow_t f) \text{ JOFFREY}) \text{ TYRION}) \\
= \text{say}_{St} (\lambda f : (St_2)_t \cdot (((\uparrow_t f) \text{ JOFFREY}) \text{ TYRION}) \text{ vac}_{(St_2)_t}) \\
= \text{say}_{St} (((\uparrow_t \text{ vac}_{(St_2)_t}) \text{ JOFFREY}) \text{ TYRION}) \\
= \text{say}_{St} (((\uparrow_t (t \epsilon (t \epsilon))) \text{ JOFFREY}) \text{ TYRION}) \\
= \text{say}_{St} (((\uparrow t \epsilon) \text{ JOFFREY}) \text{ TYRION}) \\
= \text{say}_{St} (((\lambda vst.t \cdot v \cdot s \epsilon) \text{ JOFFREY}) \text{ TYRION}) \\
= \text{say}_{St} (((\lambda st.t \cdot s \cdot t \epsilon) \text{ JOFFREY}) \text{ TYRION}) \\
= \text{say}_{St} (((\lambda t \cdot s \cdot t \epsilon) \text{ JOFFREY}) \text{ TYRION}) \\
= \text{say}_{St} \text{ TYRION} \cdot \epsilon \cdot \text{JOFFREY} \\
= \text{say}_{St} \text{ TYRION} \cdot \text{JOFFREY} \\
= \lambda s. s \text{ TYRION} \cdot \text{JOFFREY} \\
= \text{TYRION} \cdot \text{JOFFREY}
\end{align*}

Had we provided a hypothetical verb with a different phenominator, the order might have been reversed.

### 3.3 Analysis

#### 3.3.1 Overall strategy

In LCG\(_{\varphi}\), as in LCG, we simply apply functional phenoterms to their arguments with the eventual goal of obtaining strings corresponding to sentences of the language in question. However, phenogrammatical subtyping now gives us a way to distinguish between differently-“shaped” pheno functions, and thus we will be able to construct a lexical entry schema for coordination that, for example, enables the coordination of both verb phrases with verb phrases, and right node raising remnants with other right node raising remnants, while preventing the coordination of one of each.
Essentially, the phenogrammatical component of our lexical entry for *and* takes as input two conjuncts $c_1$ and $c_2$ under the image of the same phenominator $\varphi$. Here we choose to pick up the right conjunct first, which we take to be standard, although this choice is of no particular theoretical import. Next, we apply *say* to each conjunct, form the coordinate structure by concatenating the two with the string *AND* in the middle, then reapply $\varphi$ in order to create a new pheno function with the same essential structure as that of each of the conjuncts. Finally, we apply the downcasting operator $\downarrow\varphi$ in order to return to the relevant subtype, thus ensuring that for a fine-grained type $A_\varphi$, the phenotype of coordination is $A_\varphi \rightarrow A_\varphi \rightarrow A_\varphi$. Tectogrammatically, there are no surprises at the present time: the tectotype is just $B \rightarrow B \rightarrow B$ for some tectotype $B$. In keeping with the curryesque nature of LCG, this maintains the location of the tangible representation of syntax-as-word-order in the phenogrammar, rather than in the tectogrammar.

The reader should note the $\downarrow\varphi$ operator in the lexical entry for *and*. Since phenominators are defined in such a way as not to contain occurrences of the $\downarrow\varphi$ operator, the presence of this operator has as a consequence that there cannot be a phenominator whose image the pheno of the lexical entry is under. This rules out coordination of coordinating conjunctions themselves, a constraint which must in principle be stipulated in other frameworks.

**3.3.2 Revised lexical entries**

From here on out, we will suppress the semantic component of the derivations in question. Since the central point of this work exists primarily in the interface between the tectogrammar and the phenogrammar, we do not wish to muddy the waters with questions of the semantic representation of coordination. Much has been written on this topic: for discussion, see e.g. [KL14], [KL15c], [Car97], [Mor94], [Whi02], etc., and we will content
ourselves with showing how expressions take scope, a process which is obvious from the other components. We now present abbreviated lexical entries, observing that expressions now appear under the relevant phenominator, and with the semantic component suppressed.

(35) Noun Phrases
   a. aerys · the · third : St; NP
   b. arya : St; NP
   c. bran : St; NP

   likewise Catelyn, Cersei, Joffrey, Ned, Nymeria, Olenna, Robb, Sansa, Summer, Tyrion, and Tywin.

As before, noun phrases are strings in the phenogrammar, and bear tectotype NP.

(36) Intransitive Verbs / Verb Phrases
   a. (i whined) : St₁; VP
   b. (i sniveled) : St₁; VP

   Intransitive verbs and verb phrases are similarly unchanged, except for the notational convention whereby we make the phenominator and the string support explicit, to facilitate clarity to the reader in derivations. It is immediately verifiable that, e.g. (i whined) = (λvs.s · v whined) = λs. s · whined. These are unary predicates over strings (i.e. type St₁) in the phenogrammar, and have tectotype VP (i.e. NP → S).

(37) Transitive Verbs
   a. (t chastised) : St₂; TV
   b. (t hated) : St₂; TV
   c. (t killed) : St₂; TV
   d. (t slapped) : St₂; TV
Much the same is true of transitive verbs as was true of VPs, with the obvious exceptions that they are binary string predicates in the phenogrammar whose string support is under the image of the t phenominator, and have an additional corresponding NP argument in the tectogrammar, where they are of type TV, i.e. $\text{NP} \rightarrow \text{NP} \rightarrow \text{S}$.

(38) Wh-embedding Verbs

a. $\leftarrow (\text{t knew}) : \text{St}_{2}; \overline{Q} \rightarrow \text{VP}$

Wh-embedding verbs are interesting. The reader will note that they both have the same phenotype and the same phenominator as transitive verbs. This is because in both cases, they take two string arguments, which are linearized to the immediate right and the immediate left of the support, and so there is no difference between the phenogrammatical structures of the two kinds of verb. However, they exhibit different tectotypes; wh-embedding verbs are of type $\overline{Q} \rightarrow \text{VP}$ (i.e. $\overline{Q} \rightarrow \text{NP} \rightarrow \text{S}$), indicating that they require a different kind of expression as their object. Expressions of type $\overline{Q}$ will turn out to be embedded questions such as *who killed Joffrey* and *who Tyrion slapped*.

(39) Ditransitive Verbs

a. $\leftarrow (\text{d gave}) : \text{St}_{3}; \text{DV}$

Unsurprisingly, ditransitive verbs take three string arguments in the phenogrammar and their support is under the image of the d phenominator. The extend tectogrammatically from transitives (and intransitives) in the expected way, and are of type DV, i.e. $\text{NP} \rightarrow \text{NP} \rightarrow \text{NP} \rightarrow \text{S}$.

(40) Wh-words for embedded questions

a. $\leftarrow (\text{w who}) : \text{St}_{-2}; \text{VP} \rightarrow \overline{Q}$

b. $\leftarrow (\text{w what}) : \text{St}_{-2}; \text{VP} \rightarrow \overline{Q}$

Wh-words forming embedded questions are slightly more complicated. They bear the phenotype $\text{St}_{-2}$, i.e. $(\text{St} \rightarrow \text{St}) \rightarrow \text{St}$. Their string support is under the w phenominator.
\( \lambda vP.v \cdot (P \ e) \), which indicates that the first argument after the phenominator has been applied to its support is itself functional, that is, they are second-order phenogrammatical functors. We take them to be insensitive to the precise nature of the functorial arguments, and so we do not further specify the subtype of that argument. This allows for combination with any number of first order pheno predicate functors, whether the “missing” string is peripheral or otherwise. Thus embedded questions with medial gaps are possible, e.g. *who Cersei believes ___ to be plotting against her*. Tectogrammatically, they take VP arguments (i.e. NP \( \rightarrow S \)) and return embedded questions, which are of tectotype \( Q \).

(41) Determiners

a. \((\text{det} \ a) : St \rightarrow St_{-2}; N \rightarrow VP \rightarrow S\)

Determiners are exactly as before, except that we note that the phenoterm is constructed by applying the \( \text{det} \) phenominator to the string support \( a \). Quantified noun phrases end up under the image of the \( q \) phenominator:

(42) Quantified noun phrases

a. \( \vdash (q \ a \cdot \text{Coward}) : St_{-2}; VP \rightarrow S \)

b. \( \vdash (q \ a \cdot \text{Tyrant}) : St_{-2}; VP \rightarrow S \)

### 3.3.3 Canonical coordination

**String coordination**

As before, we repeat example numbering from chapter 1 in this section.

(1) [Tyrion and Sansa] hated Joffrey.

Showing that \( P_n \) obtains for the string \( \text{TYRION} \) (and respectively, \( \text{JOFFREY} \)) is trivial. We have by definition \( P_n = \lambda f : St. \exists s : St. (n \ s) = f \). Furthermore, \( n = \lambda s.s \), so \( P_n = \lambda f : St. \exists s : St. (\lambda s.s \ s) = f \), i.e. \( P_n = \lambda f : St. \exists s : St. s = f \). So \( (P_n \ \text{TYRION}) = \exists s : St. s = \text{TYRION} \), which is obviously true, and likewise for \( \text{JOFFREY} \).
In fact it is generally true that $\forall \varphi : \text{St} \to A. \forall t : \text{St}. P_\varphi (\varphi t)$, with $t$ as the relevant string support. Suppose $\varphi$ is a phenominator of type $\text{St} \to A$ and $t : \text{St}$. Then we want to show that $P_\varphi$ holds for $(\varphi t)$. By definition, $P_\varphi = \lambda f : A. \exists s : \text{St}. (\varphi s) = f$. After applying $P_\varphi$ to $(\varphi t)$ we obtain $\exists s : \text{St}. (\varphi s) = (\varphi t)$, which is obviously true with $s = t$. Since $\varphi$ and $t$ were arbitrary, $\vdash \forall \varphi : \text{St} \to A. \forall t : \text{St}. P_\varphi (\varphi t)$.

In the proofs below we elide the steps above, indicating that $P_\varphi$ holds for the terms in question by $\cdots$ followed by the relevant predicate, to aid in interpretation by the reader. Due to the fact that since the $n$ phenominator is just the identity on strings, we do not generally give lexical entries making reference to this phenominator, e.g. writing TYPRION instead of $(n \text{TYPRION})$. Nevertheless, here we will include it when necessary, writing $(n \text{TYPRION})$ instead of TYPRION, in the hopes that it will make the proof more transparent to the reader.

For the sake of brevity, we revert to the former convention once the downcasting operator $\downarrow_n$ has been applied. Furthermore we have abbreviated the proof node showing the instantiation of the lexical entry schema for and, giving it simply as AND, and we omit the typing judgment in the proof itself. We choose St for the phenotype $A$, $n$ for the phenominator $\varphi$, and NP for the tectotype $B$:

$$\text{AND} = \lambda c_2 : \text{St}_n. \lambda c_1 : \text{St}_n. \downarrow_n (n ((\text{say}_{\text{St}_n} c_1) \cdot \text{AND} \cdot (\text{say}_{\text{St}_n} c_2)))$$

$$= \lambda c_2 : \text{St}_n. \lambda c_1 : \text{St}_n. \downarrow_n (n ((\text{say}_{\text{St}_n} (\uparrow_n c_1)) \cdot \text{AND} \cdot (\text{say}_{\text{St}_n} (\uparrow_n c_2))))$$

$$= \lambda c_2 : \text{St}_n. \lambda c_1 : \text{St}_n. \downarrow_n (n ((\lambda s.s (\uparrow_n c_1)) \cdot \text{AND} \cdot (\lambda s.s (\uparrow_n c_2))))$$

$$= \lambda c_2 : \text{St}_n. \lambda c_1 : \text{St}_n. \downarrow_n (\lambda s. s ((\uparrow_n c_1) \cdot \text{AND} \cdot (\uparrow_n c_2)))$$

$$= \lambda c_2 : \text{St}_n. \lambda c_1 : \text{St}_n. \downarrow_n ((\uparrow_n c_1) \cdot \text{AND} \cdot (\uparrow_n c_2)) : \text{St}_n \to \text{St}_n ; \text{NP} \to \text{NP} \to \text{NP}$$

The analysis is simple: First, form the coordinate structure by downcasting each conjunct and applying the instantiated lexical entry for and, then upcast the result to recover a usable string. Form the verb phrase by applying hated to Joffrey. Then apply the resulting VP to the coordinate structure to obtain the sentence Tyrion and Sansa hated Joffrey.
whose phenogrammatical component is a string.\(^8\)

\[
\begin{array}{c}
\vdash \text{i (hated · Joffrey)} \\
: \text{St; VP}
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{AND} \\
: \text{St; NP} \\
\vdash \text{AND} (\downarrow_n \text{TYRION}) (\downarrow_n \text{SANS}) \\
: \text{St; NP} \\
\uparrow
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{(n TYRION)} \\
: \text{St; NP} \\
\vdash \text{AND} (\downarrow_n \text{TYRION}) \\
: \text{St; NP} \\
\vdash \text{n SANS} \\
: \text{St; NP} \\
\downarrow
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{(t HATED)} \\
: \text{St; TV} \\
\vdash \text{JOFFREY} \\
: \text{St; NP}
\end{array}
\]

\[
\begin{array}{c}
\vdash \text{AND (\downarrow_n \text{TYRION}) (\downarrow_n \text{SANS})} \\
: \text{St; NP} \\
\vdash \text{TYRION · AND · SANS · HATED · JOFFREY} \\
: \text{St; NP}
\end{array}
\]

**VP-functor coordination**

(2) Joffrey [whined and sniveled].

Verb phrase coordination is also straightforward, remembering that by “verb phrase” we mean a unary function on strings under the image of the i phenominator \(\lambda vs.s \cdot v\). As before, we make elide the lemmas showing that the candidate conjunct VPs are suitable for downcasting, but the proof is in essence the same as the one in the preceding derivation.

Here we instantiate the lexical entry schema for \(\text{and}\) with \(A = \text{St}_1\) (i.e. \(\text{St} \rightarrow \text{St}\), \(\varphi = \text{i}\), and \(B = \text{VP}\) (i.e. \(\text{NP} \rightarrow \text{S}\)):

---

\(^8\)In fact, this is more complicated than necessary. Since \(\text{St}_n = \text{St}\) and \(\uparrow_n = \lambda s.s\) by definition, the phenoterm for \(\text{and}\) could be written more simply as \(\lambda c_2 : \text{St} \cdot \lambda c_1 : \text{St} \cdot c_1 \cdot \text{AND} \cdot c_2\). We have chosen to provide the more complicated derivation here for the sake of parallelism with other examples.
\[
\text{Embedded wh-functor coordination}
\]

(3) Sansa knew [[who and what] killed Joffrey].

Coordination of embedded wh-question words may at first seem more complex than the examples we have looked at previously, but they will prove to be unproblematic. While embedded wh-question words have higher order types than any examples heretofore, this is insignificant, as they still lie under the image of a phenominator, and thus can be directly coordinated in this system.

Here we instantiate the lexical entry schema for \textit{and} with \( A = \text{St}_{-2} \) (i.e. \( (\text{St} \rightarrow \text{St}) \rightarrow \text{St} \)), \( \varphi = w \), and \( B = \text{VP} \rightarrow \overline{Q} \) (i.e. \( (\text{NP} \rightarrow \text{S}) \rightarrow \overline{Q} \)): 
AND =

\[\lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (w ((\text{say}_{\text{st}_-})_w c_1) \cdot \text{AND} \cdot (\text{say}_{\text{st}_-})_w c_2))\]

\[= \lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (w ((\text{say}_{\text{st}_-} (\uparrow_w c_1)) \cdot \text{AND} \cdot (\text{say}_{\text{st}_-} (\uparrow_w c_2))))\]

\[= \lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (w ((\text{say}_{\text{st}_-} (\uparrow_w c_1) \text{ vac}_{\text{st}_1})) \cdot \text{AND} \cdot (\text{say}_{\text{st}_-} (\uparrow_w c_2))))\]

\[= \lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (w ((\text{say}_{\text{st}_-} (\uparrow_w c_1) \lambda s.s)) \cdot \text{AND} \cdot (\text{say}_{\text{st}_-} (\uparrow_w c_2))))\]

\[= \lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (w ((\text{say}_{\text{st}_-} ((\uparrow_w c_1) \lambda s.s)) \cdot \text{AND} \cdot (\text{say}_{\text{st}_-} ((\uparrow_w c_2) \lambda s.s))))\]

\[= \lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (w ((\lambda s.s ((\uparrow_w c_1) \lambda s.s)) \cdot \text{AND} \cdot (\lambda s.s ((\uparrow_w c_2) \lambda s.s))))\]

\[= \lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (w (((\uparrow_w c_1) \lambda s.s) \cdot \text{AND} \cdot ((\uparrow_w c_2) \lambda s.s))))\]

\[= \lambda c_2 : (\text{St}_-)_w. \lambda c_1 : (\text{St}_-)_w. \downarrow_w (\lambda vP. (P \epsilon) (((\uparrow_w c_1) \lambda s.s) \cdot \text{AND} \cdot ((\uparrow_w c_2) \lambda s.s)) \cdot (P \epsilon)) \rightarrow (\text{st}_-) \rightarrow (\text{st}_-)_w : (\text{VP} \rightarrow \text{Q}) \rightarrow (\text{VP} \rightarrow \text{Q}) \rightarrow \text{VP} \rightarrow \text{Q} .\]

It will also be useful to verify that say works as expected with wh-words (using the just-calculated value for say), and the phenoterm for fine-grained who, \(\downarrow_w (w \text{ WHO})\):

\[\text{say}_{\text{st}_1 \rightarrow \text{st}_1}_w = \lambda f : (\text{st}_-)_w. (((\uparrow_w f) \lambda s.s))\]

\[\text{say}_{\text{st}_1 \rightarrow \text{st}_1}_w (\downarrow_w (w \text{ WHO}))\]

\[= \lambda f : (\text{st}_-)_w. (((\uparrow_w f) \lambda s.s)) (\downarrow_w (w \text{ WHO}))\]

\[= (\uparrow_w (\downarrow_w (w \text{ WHO}))) \lambda s.s\]

\[= (w \text{ WHO}) \lambda s.s\]

\[= (\lambda vP. v \cdot (P \epsilon) \text{ WHO}) \lambda s.s\]

\[= \lambda P. \text{ WHO} \cdot (P \epsilon) \lambda s.s\]

\[= \text{WHO} \cdot (\lambda s.s \epsilon)\]

\[= \text{WHO} \cdot \epsilon\]

\[= \text{WHO}\]
Then the rest of the derivation proceeds in the familiar manner: downcast the conjuncts, conjoin, upcast, and continue as usual, resulting in the string `sansa · knew · who · and · what · killed · joffrey`, with tectotype S, as was desired.

### 3.3.4 Noncanonical coordination

**Right node raising**

(4) [[Tyrion slapped] and [Tywin chastised]] Joffrey.

For LCG<sub>ϕ</sub>, right node raising (RNR) is not terribly mysterious, as it is just another straightforward example of functor coordination. Essentially, it looks exactly like VP-coordination, except the conjuncts in question are each looking to the right for their missing argument. We refer to these expressions as **RNR-remnants**. There are a couple of wrinkles making it marginally more complex. First, it makes use of hypothetical reasoning in order to form the remnants, and second, this choice means that we must be a little more careful about checking to ensure that the remnants are in fact under the image of the `r` phenominator λ<sub>v</sub>s · v<sub>s</sub>. Showing the latter is nevertheless quite easy, and the proof proceeds as follows. We ask the reader to take as given that an exemplary RNR remnant has the following phenogrammatical form (this is shown in the full proof): λ<sub>s</sub> : St · tywin · chastised · s<sub>tywin</sub>. If we choose as string support the string `tywin · chastised`, then it is clear that `(r tywin ·
chastised) = λs : St. tywin · chastised · s. By the lemma proved earlier, then, \( P \) obtains for \( \lambda s : St. tywin · chastised · s \) (and will obtain likewise for the second conjunct). As always, we elide this in the proof below.

The reader will note that for the sake of legibility, we provide separate proofs L1 and L2 of the derivations of each of the conjuncts, and that these are to be inserted in the main proof at the labeled point. We instantiate the lexical entry schema for and in the following way: \( A = St_1 \), \( \varphi = r \), and \( B = VP \). This results in the following lexical entry:

\[
\begin{align*}
\text{AND} &= \\
\lambda c_2 : (St_1), \lambda c_1 : (St_1), & \downarrow, (r ((\text{say}_{(St_1)} \cdot c_1) \cdot \text{AND} \cdot (\text{say}_{(St_1)} \cdot c_2))) \\
= \lambda c_2 : (St_1), & \downarrow, (r ((\text{say}_{St_1} ((\uparrow, c_1)) \cdot \text{AND} \cdot (\text{say}_{St_1} ((\uparrow, c_2))))) \\
= \lambda c_2 : (St_1), & \downarrow, (r ((\text{say}_{St_1} ((\uparrow, c_1) \epsilon) \cdot \text{AND} \cdot (\text{say}_{St_1} ((\uparrow, c_2) \epsilon))))
\end{align*}
\]

\[
\begin{align*}
\lambda c_2 : (St_1), & \downarrow, (r ((\lambda s \cdot ((\uparrow, c_1) \epsilon) \cdot \text{AND} \cdot ((\uparrow, c_2) \epsilon)))) \\
= \lambda c_2 : (St_1), & \downarrow, (\lambda vs \cdot s ((\uparrow, c_1) \epsilon) \cdot \text{AND} \cdot ((\uparrow, c_2) \epsilon)) \\
= \lambda c_2 : (St_1), & \downarrow, (\lambda t \cdot t ((\uparrow, c_1) \epsilon) \cdot \text{AND} \cdot ((\uparrow, c_2) \epsilon)) \\
: St_r \rightarrow St_r \rightarrow St_r; VP \rightarrow VP \rightarrow VP
\end{align*}
\]

L1.

\[
\begin{array}{c}
\vdash \lambda st \cdot t \cdot \text{chastised} \cdot s \\
: St_2; TV \\
\hline
x : St; NP \vdash x : St; NP \\
x : St; NP \vdash \lambda t \cdot \text{chastised} \cdot x \\
: St_1; VP \\
\hline
x : St; NP \vdash tywin \cdot \text{chastised} \cdot x \\
: St; NP \\
\hline
\vdash tywin \cdot \text{chastised} \cdot x \\
: St, S \\
\hline
\vdash \lambda x : St. tywin \cdot \text{chastised} \cdot x \\
: St_1; VP \\
\hline
\vdash \downarrow, \lambda x : St. tywin \cdot \text{chastised} \cdot x \\
: (St_1),
\end{array}
\]

L2.
The reader will note in L1 and L2 that the main difference between this derivation and the ones preceding is the use of hypothetical reasoning to create the conjuncts in question. Each transitive verb is first applied to a hypothesis corresponding to its object argument, then to its subject, and the hypothesis is subsequently withdrawn in order to form the RNR-remnant conjunct.

From this point on everything proceeds as normal: the conjuncts are properly subtyped to be conjoined, so we apply and to each conjunct in turn to create the coordinated RNR-remnant, which is then upcast in order to restore its ability to apply to its object Joffrey, resulting in the phenogrammatical string TYRION-SLAPPED-AND-TYWIN-CHASTISED-JOFFREY which is asserted to be a sentence in the tectogrammar.

**Argument cluster coordination**

So-called *argument cluster coordination* (henceforth ACC) takes place when each conjunct of a coordinate structure appears to be a sequence of complements of the same verb, as in the following (where *Summer* and *Nymeria* are understood to be dire wolves):
(5) Ned gave [[Bran Summer] and [Arya Nymeria]].

Such examples are traditionally problematic from the perspective of MGG, since the expressions *Bran Summer* and *Arya Nymeria* do not form constituents in the traditional (phrase-structural) sense. Categorial grammars with directional implications have provided analyses of such examples by treating the conjuncts as having category DV\VP, so that the coordinate structure resulting from their conjunction forms a VP by combining with a DV to its left. In [Dow88]'s CCG analysis, for which hypothetical reasoning is unavailable, the argument clusters are formed by giving the indirect and direct objects the raised types DV\TV and TV\VP respectively, and then combining them via right-to-left function composition [Dow88]. Here VP, TV, and DV are defined not in terms of $\rightarrow$ as in LCG, but rather as the customary directional categories NP\S, (NP\S)/NP, and ((NP\S)/NP)/NP, respectively. In the case of Lambek categorial grammars (see e.g. [Car97]), the clusters can be formed in a more straightforward fashion, by combining a hypothetical DV with its two (unraised) NP objects and then withdrawing the hypothesis. Either way, directionality is centrally involved, because the complement clusters end up with the (directional) category DV\VP = (((NP\S)/NP)/NP)/(NP\S).

As a consequence, as with other cases of functor coordination, ACC is beyond the reach of curryesque frameworks unless they are augmented with some mechanism for making reference to direction of combination. In extended Lambek CGs like HTLG, the Lambek-style analysis can be imported wholesale, as it is in [KL13]). In LCG$\varphi$, however, where the tectotypes are all nondirectional, this option is not available. Instead, we will show that a complement cluster like *Bran Summer* is not just any old DV $\rightarrow$ VP, but moreover has the crucial property that its phenomenogrammatical component lies in the image of a certain phenomenator $\text{a}$, to be defined immediately below. This will make use of a phenomenator which is somewhat different than the ones seen previously, but it will follow from this that ACC can be straightforwardly analyzed by an instance of the coordination schema already proposed.
The $a$ phenominator is by necessity more complex than any phenominator we have previously seen, since in ACC constructions, we must make crucial reference to the fact that the verb whose arguments the construction itself is comprised of must have a particular shape, namely, that of a ditransitive. So the $a$ phenominator itself must encode this fact in the subtype of its higher-order functional argument. We define the ACC phenominator $a$ as follows:

$$a = \text{def} \lambda v : \text{St.} \lambda p : (\text{St}_3)_d \cdot \lambda s : \text{St.} s \cdot (\text{say}_{(\text{St}_3)_d} p) \cdot v \text{ (argument cluster coordination)}$$

As an immediate consequence of the definition of $\text{say}$, we have

$$a = \lambda v : \text{St.} \lambda p : (\text{St}_3)_d \cdot \lambda s : \text{St.} s \cdot (\uparrow_d p \in \epsilon \epsilon) \cdot v$$

We find it both spatially and presentationally useful to prove lemmas for intuitively identifiable subparts of the full proof, which we then provide with numerical indices to be inserted in the full proof. We make use of this convention here for the two conjuncts in the coordinate structure (L3 and L4), the insertion of the lexical entry for and (L8), and for the subtyping of the verb (L7).

First, we derive the desired structure for the first conjunct, the cluster Bran Summer. The essential strategy is to hypothesize a ditransitive verb, apply it to its two arguments, withdraw the hypothesis, and then subtype according to the result, which will be under the argument cluster coordination phenominator $a$. Here we see the first practical instance of a phenominator which itself makes reference to fine-grained types, since $a$ invokes the type $(\text{St}_3)_d$. This will be essential, since we will need the eventual coordinate structure to select for a ditransitive verb specifically, and not a general ternary string function.
L4 \quad \vdash_a (\lambda h : (\text{St}_3)_a. \uparrow_d h \text{ ARYA NYMERIA}) : ((\text{St}_3)_d \to \text{St}_1)_a; \text{DV} \to \text{VP}

The formation of the second conjunct, the cluster *Arya Nymeria*, proceeds exactly as the derivation given in L3, changing the names of the constants *bran* and *summer* to *arya* and *nymeria*, respectively.

**Subtyping justification for the conjuncts**

In order to coordinate the clusters in question, we need to show that they do in fact lie in the image of the relevant phenominator, namely *a*. In particular, we want to show that

\[
L5 \quad \vdash P_a (\lambda h : (\text{St}_3)_a. \uparrow_d h \text{ BRAN SUMMER})
\]

Suppose that we have \( h : (\text{St}_3)_d \). Now, since \( h = (\downarrow_d (\uparrow_d h)) \), it is easy to see that \( \uparrow_d h \) is in the image of \( d \). To put it another way, \( P_a \) holds of \( \uparrow_d h \). So by the definition of \( P_a \),

\[
\vdash \exists v : \text{St}. \uparrow_d h = d \cdot v.
\]

So we can ask, what is \( v \)? By definition, it is the string support of \( \uparrow_d h \), and we can therefore obtain it by taking \( \text{say}_{\text{St}_3}(\uparrow_d h) \). After working through the definition of \( \text{say} \), we are left with the support \( (\uparrow_d h \cdot e \cdot e) \). So \( v = (\uparrow_d h \cdot e \cdot e) \), and thus

\[
\vdash d \cdot v = d (\uparrow_d h \cdot e \cdot e) = \lambda stu : \text{St.} u \cdot (\uparrow_d h \cdot e \cdot e) \cdot s \cdot t.
\]

Now, if we apply the injection of \( h \) to the two arguments we wish to create the cluster from, we can see that

\[
\vdash \uparrow_d h \text{ BRAN SUMMER} = \lambda stu : \text{St.} u \cdot (\uparrow_d h \cdot e \cdot e) \cdot s \cdot t \text{ BRAN SUMMER}.
\]

After reduction, we are left with the following:

\[
\vdash \uparrow_d h \text{ BRAN SUMMER} = \lambda u : \text{St.} u \cdot (\uparrow_d h \cdot e \cdot e) \cdot \text{BRAN} \cdot \text{SUMMER}.
\]

Furthermore, by binding our original hypothesis \( h \) in each side, we see that
\[ \vdash \lambda h : (St_3)_d. \uparrow_d h \text{ BRAN SUMMER} = \lambda h : (St_3)_d. \lambda u : St. u \cdot (\uparrow_d h \epsilon \epsilon) \cdot \text{BRAN} \cdot \text{SUMMER}. \]

So now we are in a position to ask the question we were originally curious about with respect to ACC subtyping: does \( P_a \) hold of \( \lambda h : (St_3)_d. \uparrow_d h \text{ BRAN SUMMER} \)? As a consequence of the previous line of reasoning, this is the same as asking whether \( P_a \) holds of \( \lambda h : (St_3)_d. \lambda u : St. u \cdot (\uparrow_d h \epsilon \epsilon) \cdot \text{BRAN} \cdot \text{SUMMER} \).

By examining the definition of \( P \) more precisely, we can see that we are asking about the truth of the following:

\[ \vdash \exists v : St. \lambda h : (St_3)_d. \lambda u : St. u \cdot (\uparrow_d h \epsilon \epsilon) \cdot \text{BRAN} \cdot \text{SUMMER} = \lambda p : (St_3)_d. \lambda s : St. s \cdot (\uparrow_d \epsilon \epsilon) \cdot v. \]

Since \( a \) is defined as \( \lambda v : St. \lambda p : (St_3)_d. \lambda s : St. s \cdot (\uparrow_d \epsilon \epsilon) \cdot v \), this amounts to asking whether

\[ \vdash \exists v : St. \lambda h : (St_3)_d. \lambda u : St. u \cdot (\uparrow_d h \epsilon \epsilon) \cdot \text{BRAN} \cdot \text{SUMMER} = \lambda p : (St_3)_d. \lambda s : St. s \cdot (\uparrow_d \epsilon \epsilon) \cdot v. \]

After \( \alpha \)-conversion, the answer is yes, with \( v = \text{BRAN} \cdot \text{SUMMER} \), so

\[ \vdash P_a (\lambda h : (St_3)_d. \uparrow_d h \text{ BRAN SUMMER}) \]

and the subtyping is justified.

L6 \( \vdash P_a (\lambda h : (St_3)_d. \uparrow_d h \text{ ARYA NYMERIA}) \)

As in L5, mutatis mutandis.

**Subtyping of gave**

Finally, we derive the subtyping of the ditransitive verb *gave* as being under image of the d denominator:

\[
\begin{align*}
L7 & \quad \vdash d \text{ GAVE} : St_3; DV \\
& \quad \vdash P_a (d \text{ GAVE}) \\
& \quad \vdash \downarrow_d (d \text{ GAVE}) : (St_3)_d; DV
\end{align*}
\]

81
As for the justification of the second premiss, it follows immediately from the definition of $P$ that \( P_d \) (\( d \) gave).

Owing to the complexity of the structure of the conjunct clusters, the instantiation of our coordination lexical entry is similarly complex, with $B = DV \rightarrow VP$, $A = (St_3)_d \rightarrow St_1$, and $\varphi = a$:

\[
\begin{align*}
\vdash \lambda c_2 c_1 : ((St_3)_d \rightarrow St_1)_a. & \downarrow a ((\text{say}(St_3)_d \rightarrow St_1) \rightarrow \text{say}(St_3)_d \rightarrow St_1 c_2)) \\
& \downarrow ((St_3)_d \rightarrow St_1)_a \rightarrow ((St_3)_d \rightarrow St_1)_a \rightarrow ((St_3)_d \rightarrow St_1)_a \\
& (DV \rightarrow VP) \rightarrow (DV \rightarrow VP) \rightarrow (DV \rightarrow VP)
\end{align*}
\]

It is useful to see exactly how the recursive definition for \textit{say} works out for the type of the conjuncts in question. Perhaps the most central insight to be gleaned from this comes from asking the question of how exactly one “says” a vacuous verb. The answer is that while you can’t “say” the verb itself, you could “say” whatever arguments it has in the order that the verb specifies. This is accomplished by taking the denominator associated with the verb in question (here, $d$), and applying it to the empty string, in order to create a function that places the verbal string arguments in the correct relative location.

\[
\begin{align*}
\text{say}(St_3)_d \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}(St_3)_d \rightarrow St_1 (\uparrow a p) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p \text{ vac}(St_3)_d) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p (\downarrow d (\epsilon))) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p (\downarrow d (\lambda v stu. u \cdot v \cdot s \cdot t \epsilon))) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p (\downarrow 4d (\lambda stu. u \cdot s \cdot t))) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p (\downarrow 4d (\lambda stu. t \cdot s \cdot u))) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p (\downarrow 4d (\lambda stu. u \cdot s \cdot t \text{ vac}_{St}))) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p (\downarrow 4d (\lambda stu. u \cdot s \cdot t) \epsilon)) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & \text{say}_{St_1} (\uparrow a p (\downarrow 4d (\lambda stu. u \cdot s \cdot t) \epsilon)) \\
= \lambda p : ((St_3)_d \rightarrow St_1)_a. & (\uparrow a p (\downarrow 4d (\lambda stu. u \cdot s \cdot t) \epsilon))
\end{align*}
\]
It is easy to see, then, that the lexical entry for ACC and boils down to the following:

\[ \lambda c_2 \downarrow a (a ((\downarrow_a c_2 (\downarrow a \lambda u \cdot s \cdot t) \epsilon) \cdot \text{AND} \cdot (\downarrow_a c_1 (\downarrow a \lambda u \cdot s \cdot t) \epsilon))) \]

\[ \Rightarrow ((St_3)_d \rightarrow St_1)_a \rightarrow ((St_3)_d \rightarrow St_1)_a \rightarrow ((St_3)_d \rightarrow St_1)_a \]

\[ \Rightarrow (DV \rightarrow VP) \rightarrow (DV \rightarrow VP) \rightarrow (DV \rightarrow VP) \]

For the sake of brevity, we stipulate that the conjunct variable \( q \) has type \( ((St_3)_d \rightarrow St_1)_a \), and we will omit the type annotation from the derivation below. We will sometimes add line breaks to the signs below to improve legibility, and we suggest that the reader read \( : \) as “has phenotype” and \( ; \) as “has tectotype”. The reader should note the inference following the inclusion of L4, wherein the phenominator \( a \) appears in its explicit form, applied to the completed coordinate structure, rather than in its previous abbreviated form.

\[
\begin{array}{c|c}
L8 & L3 \\
\hline
\Rightarrow \lambda q. \downarrow a (a (\text{BRAN} \cdot \text{SUMMER} \cdot \text{AND} \cdot (\downarrow a \lambda u \cdot s \cdot t) \epsilon)) \\
\Rightarrow ((St_3)_d \rightarrow St_1)_a \rightarrow ((St_3)_d \rightarrow St_1)_a \\
\Rightarrow (DV \rightarrow VP) \rightarrow (DV \rightarrow VP) \\
\Rightarrow (\lambda p : (St_3)_d. \lambda s. (\downarrow a \lambda p \epsilon \epsilon) \cdot \text{BRAN} \cdot \text{SUMMER} \cdot \text{AND} \cdot \text{ARYA} \cdot \text{NYMERIA}) \\
\Rightarrow ((St_3)_d \rightarrow St_1)_a; DV \rightarrow VP \\
\Rightarrow (\lambda p : (St_3)_d. \lambda s. (\downarrow a \lambda p \epsilon \epsilon) \cdot \text{BRAN} \cdot \text{SUMMER} \cdot \text{AND} \cdot \text{ARYA} \cdot \text{NYMERIA}) \uparrow \\
\Rightarrow (\lambda s. s \cdot \text{GAVE} \cdot \text{BRAN} \cdot \text{SUMMER} \cdot \text{AND} \cdot \text{ARYA} \cdot \text{NYMERIA} : St; VP) \\
\Rightarrow \text{ned} : St; NP \\
\Rightarrow \text{ned} \cdot \text{gave} \cdot \text{BRAN} \cdot \text{SUMMER} \cdot \text{AND} \cdot \text{ARYA} \cdot \text{NYMERIA} : St; S \\
\Rightarrow \text{ned} \cdot \text{gave} \cdot \text{BRAN} \cdot \text{SUMMER} \cdot \text{AND} \cdot \text{ARYA} \cdot \text{NYMERIA} : St; S
\end{array}
\]

In summary: we first apply the sign for and to each conjunct. This results in an expression “missing” both its ditransitive verb (a ternary function over strings which is under the image of the \( d \) phenominator) and its NP subject (a string). This entire expression is under the \( a \) phenominator, meaning that to make use of it, we pass it through the \( \uparrow \) rule, embedding it within its supertype. Now we can apply it first to the ditransitive verb gave and the NP Ned, in order to obtain the complete expression Ned gave Bran Summer and Arya Nymeria, which is asserted to be a phenogrammatical string, and a sentence in the tectogrammar, as was desired.
3.4 Discussion

We have provided an array of coordination data which LCG can be extended to LCG_ϕ in order to account for. We have evaluated our strategy with respect to other kinds of categorial grammar, and found that it is possible to extend the empirical coverage of cur-ryesque grammars in order to achieve similar results to the coverage of Lambek categorial grammars. We have defined phenominators, certain kinds of linear combinators that we use in conjunction with Lambek and Scott-style subtyping in order to differentiate between expressions of otherwise like phenotype and, for that matter, like tectotype. We provide rules of upcasting and downcasting in order to manipulate subtypes, and we have defined the function `say` and its associated vacuities to enable the extraction of string supports from LCG_ϕ phenoterms.

We have given lexical entries for the fragment given here, including a lexical entry schema for `and` which is general enough to be sufficient to analyze the data provided. Finally, we have provided derivations for each of the examples given.
Chapter 4

Predicatives and Unlike Category Coordination in LCG

4.1 Overview and Data

Traditional accounts of coordination in English posit “Wasow’s Generalization”, named so in [PZ86] after [SGWW85]: that coordination is a phenomenon whereby like constituents are conjoined in such a manner as to serve in the syntactic position where either one could serve on its own. Categorial grammars, in particular [Ste85] and [Dow88] have had much success analyzing what we refer to as “noncanonical” coordinate constructions.

One of the ways in which a coordinate structure can fail to be canonical is by having conjuncts which are not (or at least do not appear to be) of the same syntactic category. This is referred to variously as “coordination of unlikes”, “unlike category coordination”, and is an instance of the more general phenomenon of “functor neutrality”, to be discussed shortly. We will typically use unlike category coordination, abbreviated UCC, to refer to the phenomenon. Here again, logical categorial grammars have had success providing an analysis, typically couched in terms of the disjunction operator $\lor$ ([Bay96], [Mor94]).

In [PH03] Pollard and Hana point out a fundamental misunderstanding with this analysis, and make steps towards remediying it. Unfortunately, their analysis does not go quite far enough, as it is expressed in a theory (Higher Order grammar) whose ability to account for facts about linear order was somewhat impoverished.

We have previously discussed (in Chapter 3 and [Wor14]) how to restore a notion of “directionality” to a “Curryesque” grammar formalism based on a linear, rather than a
bilinear system. In this chapter we present a way to augment the tectogrammatical component of Linear Categorial Grammar (LCG) with a special kind of monad type as studied in [Mog91] and [BBdP98] among others, extending and correcting the analysis in [PH03]. While this connective is not native to linear logic\(^1\),\(^2\) we show how it may be added in a principled manner, a move anticipated in [Mor94]. In short, the augmentation of a linear type system with nonlinear connectives is an empirically sensible maneuver, since coordinate structures are by their very nature nonlinear. This nonlinearity manifests itself in two ways: first, in the use of two conjoined expressions where one would suffice; and second, in the iterated case, where the first conjunct consists of a list of similar expressions.

It will turn out that both of these descriptive generalizations can be analyzed using the same fundamental concept: that of the monad, albeit three different monads, which we will refer to as the coordinate structure monad (or coordination monad), the list monad (or string monad), and the (finite) multiset monad, though this chapter will only discuss the first of these. Discussions of the other two follow in chapter 5.

Monads have been known in category theory under that name as far back as [ML78], although the concept originated in [God58]. It has taken on at least two other lives. In computer science, the term typically refers to strong monads (to be defined later), and they are generally given in the manner of a Kleisli triple, though it remains a close companion to the categorical monad. In formal logic, monads can be seen as a kind of modality resembling modal possibility\(^3\), and, in the case of strong monads, as the lesser-known lax modality \(\Box\), which shares aspects of both possibility and necessity, and is typically described as representing “true under some constraints” [FM97], [BBdP98]. In order to provide an analysis of unlike category coordination, the coordinate structure monad will be of particular

\footnote{Linear logic \textbf{does} provide a monad type constructor, the “why not” connective \(?\), used together with its dual \(!\) to reintroduce the structural rules of contraction and weakening in a principled way. We have in mind a monad with a different purpose, and so we propose to add such a type constructor more generally.}

\footnote{The idea seems to have been present in [Tro92], where a similar construction called a “closure” is described.}

\footnote{In fact, a very similar connective was proposed as a modal possibility connective independent of necessity in [Cur57].}
use.

The first section provides an overview of the data in question. In section 4.2, we sketch our analytical strategy and discuss previous analyses. Section 4.3 provides a technical primer to LCG, and introduces coordination monad types. In section 4.4, we give lexical entries and LCG proofs for selected examples from the text. Section 4.5 discusses the results, and additional proofs appear in the final section.

4.1.1 Predicatives

The copula is a quite complicated piece of the English language, and so we will be forced to make a number of simplifying assumptions about it, lest we lose sight of the central point of this dissertation.

Copular predicatives

(43)  a. Joffrey is evil.
     b. Joffrey is Aerys III.
     c. Joffrey is a tyrant.
     d. Joffrey is on the iron throne.

As illustrated by the examples above, a great number of different types of expressions can serve as copular complements. After [Whi02] and others, we refer to the ability of a functor (in this case a verb) to combine with multiple kinds of expressions, and crucially, coordinate structures consisting of combinations of these, as functor neutrality. By contrast, the ability of coordinated verb with differing dependencies to combine with a single argument fulfilling each of those dependencies is referred to as argument neutrality. Though not obviously relevant for English coordination, argument neutrality will be discussed briefly in chapter 4.4.4. We refer to the kinds of complements selected for by the copula as predicatives.
For the present discussion, we limit ourselves to non-verbal predicatives. It is possible to extend the analysis given here in such a way that verbal copular complements (that is, participial phrases and passive phrases) are accounted for. So we will make the simplifying assumption that the copula selects for: predicative adjective phrases (43a), predicative nominals including names (43b) and quantified noun phrases (43c)\(^4\), and predicative prepositional phrases (43d)\(^5\), though it remains neutral with respect to which argument(s) it receives.

In all of the above examples, we have a third person singular subject. Since the English copula exhibits a host of agreement phenomena, and since it is our intent to discuss coordination, and not agreement, we will omit the rest of the distinctions among copular forms and their subjects from discussion, noting in passing that this simplification also results in the overgeneration of sentences like *Joffrey and Tyrion is a tyrant.*

**Other predicatives**

The copula is by no means the only verb which is capable of exhibiting neutrality with respect to predicatives. We will initially address the verb *became,* which, unlike the copula, is pleasingly free from large agreement paradigms. We will additionally consider examples like the following:

\[(44)\]

a. Joffrey became evil.

b. Joffrey became Aerys III.

c. Joffrey became a tyrant.

d. * Joffrey became on the iron throne.

\(^4\)It is possible to analyze the selectional properties of the copula without making specific reference to two different kinds of noun phrases, and we do so here only as a convenience.

\(^5\)as opposed to prepositional phrases headed by "case-marking" prepositions
4.1.2 Coordinate structures composed of predicatives

In addition to being neutral with respect to their argument selection, verbs of predication exhibit one other interesting property: namely, they remain neutral with respect to their arguments even when said arguments exist within a coordinate structure. That is, verbs of predication seem to be entirely satisfied by receiving any of their arguments individually, or in any kind of mixed coordinate structure, as long as each argument comprising the coordination is permissible to the verb (example numbering repeated from chapter 1):

(6) Joffrey is evil and a tyrant.
(7) Joffrey is on the iron throne and a tyrant.
(8) Joffrey is evil and on the iron throne.
(9) Joffrey is evil and a tyrant and on the iron throne.
(10) Joffrey became evil and a tyrant.

This is an instance of functor neutrality, a subtype of what is referred to as unlike category coordination (UCC), since the elements of the coordinate structure are dissimilar categories. The phenomenon of UCC in general is somewhat broader, additionally including for example cases of argument neutrality along the lines of those discussed in section 4.4.4. This flies in the face of standard analyses of coordination such as those originating in [Ste85] and [Dow88], which hold that only “like” categories may be coordinated. In addition to the mystery presented by UCC itself, these phenomena fly in the face of traditional wisdom by suggesting that in some cases, the category of coordinate structures composed of particular categories is different, and in some way larger, than the category of the structures which comprise the coordinate structure.
4.2 Analytical strategy and comparison with previous approaches

The central question is: what kind of thing is evil and a tyrant? It is not a noun phrase; it is not an adjective phrase. It is neither; instead, the obvious answer is that it is a coordinate structure consisting of a noun phrase and an adjective phrase. So what type are we to give this?

4.2.1 Bayer, Johnson, and Morrill

Since at least [BJ95] and [Bay96] the standard categorial account of UCC phenomena makes use of the disjunction type constructor \( \lor \), which, as we will see, is mostly the right idea, though [PH03] correctly notes that there is an important foundational problem with this analysis.

In a setting where the disjunction type is interpreted as the disjoint union of sets (as is often the case with \( \lor \) and related connectives), then a UCC construction like evil and a coward is typed as being in the disjoint union of two sets. That is, the type given to the entire expression asserts that it must have originated in either the set of predicative APs, or in the set of predicative NPs. But evil and a coward is not either one of these; it belongs to a slightly larger category, the category of things which are APs or NPs or coordinate structures made up of APs and / or NPs. So the category assignment given to such constructions is not quite correct.

Morrill essentially adopts the analysis of Bayer and Johnson in both [Mor94] and [Mor10], and consequently inherits from them the same deficiency, albeit in a proof-theoretic rather than a model-theoretic setting. The omission of genuine proof terms from what might be deemed the tectogrammar\(^6\) at this point obscures an important fact about the use of \( \lor \) with respect to coordination. Notably, the only way for a given expression to be of type

\(^6\)The systems described in this section do not in general distinguish between tectogrammar and phenogrammar, and so have a singular, syntactocentric type system.
A ∨ B is for it to have originated as type A or type B (in particular, the rules of proof typically invoke canonical injections \texttt{inl} : A → (A ∨ B) and \texttt{inr} : B → (A ∨ B)). As was the case with the Bayer and Johnson typing, the entire coordinate structure did not originate in either category.

4.2.2 Whitman

In [Whi02], Whitman proposes a slightly different solution to the problem by turning the issue on its head. This analysis is based on an a labelled extension of the Lambek Calculus after [Mor94], and so does not extend directly to LCG, since LCG has a more complex phenogrammar, and a simplified tectogrammar. For the sake of comparison, we will translate Whitman’s \( \vdash \) and / connectives as \( \rightarrow \), and then we may at least see how the tectogrammatical component of LCG squares up. Though Whitman considers modeling functor neutrality using disjoint union types as the argument, he dismisses this on grounds that this system suffers from incompleteness in the presence of both \( \wedge \) and \( \vee \), on a set-theoretic interpretation of the connectives. This is only a problem for the set-theoretic interpretation, however, and not necessarily for systems using these connectives.

Since he cannot dispense with additive conjunctive types for exterior reasons, he elects to dismiss the disjunction connective, and use conjunctive types over the functors themselves. Those expressions that would typically be assigned category \((A ∨ B) \rightarrow C\), again, using \( \rightarrow \) instead of one of the directional Lambek slashes, are instead assigned the equivalent category \((A \rightarrow C) \wedge (B \rightarrow C)\). These category assignments are interderivable in propositional intuitionistic linear logic. So in the simple (non-coordinate) case, customary rules of \( \wedge \)-elimination will allow us to choose the functor we want. In the UCC case, Whitman type raises over the relevant functor category (e.g. expressions of type A are raised to \((A \rightarrow C) \rightarrow C\), likewise for B, and then he uses the theorem of antecedent weakening to derive the category \( ((A \rightarrow C) \wedge (B \rightarrow C)) \rightarrow C\) for both conjuncts. So they receive like types, albeit at the expense of becoming necessarily type-raised.
It is interesting to note that “type raising” is essentially another name for the embedding of so-called “direct types” into “continuation types”, and that continuations can be modeled by a strong monad. In fact, in a system which has both additive conjunction (typically written \( \land \) or \&) and additive disjunction (typically written \( \lor \) or \( \oplus \)), Whitman’s category assignment (translated into linear logic) for type-raised predicatives \(((A \to C) \& (B \to C)) \to C\) is provably equivalent to our type \( C \), if \( C \) were to be defined as \( (A \to C) \to C \), that is, an \( A \)-continuation with answer type \( C \), represented by a strong monad. If we wish to maintain this strategy, we need to do one of two things: either use multiple continuation monads (for each answer type \( C \), as above), or work in a fragment of second-order linear logic (that is, intuitionistic linear logic, together with explicit type polymorphism), allowing us to define \( C \) as \( \forall \alpha. (A \to \alpha) \to \alpha \), with \( \alpha \) a type variable.

So in essence Whitman’s solution is quite similar to the one presented here, albeit expressed in a more oblique fashion. Since we do not presently wish to equate coordinate structures to continuations, and since we do not use the tectogrammar as the primary source of “directionality”, the “coordinate structure monad” analysis is simpler and somewhat more general, in that the strong monad constructor \( \circ \) does not explicitly create continuations in any component of the grammar, though it could with the addition of one definition, retrieving an analysis analogous to that in [Whi02] if desired.

4.2.3 Pollard and Hana

In essence, the analysis presented in [PH03] extends that of Bayer, Johnson, and Morrill in the correct way, correctly noting the issue with the proof theory and interpretation of whatever version of disjunction used. Nevertheless, it is not free from issues. Most notably, the lack of a fine-grained theory of the phenogrammar of directionality makes it unclear exactly how expressions are to be constructed. As far as the tectogrammar is concerned, Pollard and Hana’s analysis requires them to make a global change to functor structure. Instead of simply taking \( A \) as an argument (for any \( A \)), functors need to take \( \text{GEN}[A] \),
a coordinate structure of As. This is a significant lexical revision; every single functor must have all of its arguments “painted up” with GEN[], and every single time a functor combines with an argument, the argument must be injected into the generalized type. So this analysis mandates both a great number of extra proof steps, and somewhat verbose syntactic category assignments.

Moreover, [PH03] fails to generalize to the case of functor coordination. While it is obvious that in order to coordinate functors, they must be injected into a GEN[] type, this says nothing about their ability to subsequently combine with their arguments. Functor coordination presents another problem for Pollard and Hana, in that their system, being curryesque (albeit with higher order logic as the basis for the tectogrammar) suffers from exactly the same problems most curryesque grammars do, with respect to losing any concept of directionality. So the system outlined in [PH03] is untenable both tectogrammatically and phenogrammatically.

Nevertheless, our analysis is an extension of their analysis, and once we recognize GEN[] as a strong monad, and give suitable inference rules describing the behavior of that connective, major lexical revision will be unnecessary, since the functoriality property of the monad which we will give as a lemma will guarantee that any functor is able to combine with a coordinated argument. Additionally, the problem of functor coordination will be solved, since the property of cotensorial strength, again to be provided as a lemma, allows us to “push” the monad down on to return type of a coordinated functor, restoring its ability to combine with its arguments. Furthermore, since LCG has fine-grained phenotypes, the directionality issue is skirted entirely as well.

4.2.4 LCG

We adopt an analysis in the spirit of [PH03], albeit couched in a tectogrammar based on linear logic, rather than HOL.
So we will need to take a slightly different tack. Most of what is required are modifications to the tectogrammar and to a certain extent, the semantics. The phenogrammar of unlike category coordination should remain essentially similar to the phenogrammar of like category coordination.

In [PH03], this constructor is written GEN[], but we suggest that it is more recognizable as $\bigcirc$, the “sometimes” modality from lax logic [FM97], and furthermore, the general monad type constructor from the body of literature originating with [Mog91]. Interestingly, such a connective was briefly considered by Curry in [Cur57], which was based on lectures delivered in 1948. So our grammar is doubly curryesque.

Monads generally seem to have exactly the right properties for coordination. Intuitively, since this type should include both $A$s and coordinate structures composed of $A$s, we should have a mapping from $A$ to $\bigcirc A$, and this is represented as the introduction rule (itself the proof-theoretic incarnation of the unit ($\eta$) natural transformation of a categorical monad).

Additionally, we note that it is entirely possible to have coordinate structures built out of other coordinate structures, as is the case with coordinations like Tyrion and Cersei and Joffrey and Tywin, and that these are interchangeable with “simpler” two-item coordinate structures, and so we would like to be able to give them the same category. But here again, the categorical monad provides exactly what we need: the $\mu$ natural transformation (sometimes called “multiplication” or “idempotence”), mapping $\bigcirc \bigcirc A$ to $\bigcirc A$. In effect, this rule just says “a coordinate structure made out of coordinate structures is itself a coordinate structure”.

The change from GEN[] to $\bigcirc$ will provide additional benefits, as it suggests a way out of the problems with the analysis in [PH03] alluded to in section 4.2.3. The shift to using $\bigcirc$ specifically provides as theorems two crucial properties addressing the outstanding issues of lexical revision and functor coordination. First, $\bigcirc$ gives us the ability to “retrofit” lexical entries on the fly in order to allow functors to accept coordinated arguments, so no global revision of the lexicon is necessary. Second, coordinated functors will still be able to accept
their normal arguments, as long as the information that coordinate structures are involved is maintained in the return type of the functor. Finally, we will need to revise the lexical entries of any coordinate functors to ensure that, tectogrammatically, they take as arguments not expressions of type $A$, but of $\circ A$, and so should be given the type $\circ A \rightarrow \circ A \rightarrow \circ A$.

### 4.3 Technicalia

The type theory underlying the tectogrammatical component of LCG is based on what we refer to (temporarily) as the **implicational fragment** of linear logic (LL) [Gir87]. Since the interactions of the various connectives of linear logic are well-understood, it is to that framework that we will initially turn when it comes to enriching the grammar formalism with connectives that behave in the manner desired to provide an analysis of unlike category coordination. Linear logic is generally described as being “resource-sensitive”, owing to the lack of the structural rules of weakening and contraction. Resource sensitivity is an attractive notion, theoretically, since it allows us to describe processes of resource production, consumption, and combination in a manner which is agnostic about precisely how resources are combined. Certain problems which have been historically tricky for Lambek categorial grammars (medial extraction, quantifier scope, etc.) are easily handled by LCG.

Grammar rules take the form of derivational rules which generate triples called **signs**, and they bind together the three logics so that they operate concurrently. While the invocation of a grammar rule might simply be, say, point-wise application, the ramifications for the three systems can in principle be different; one can imagine expressions which exhibit type asymmetry in various ways, as we will see. For a quick example, we will note that expressions may be strings in the phenogrammar, but have multiple disjunctive types in the tectogrammar. Specifically, the paired **Inj** rules will allow the same phenogrammatical string, say, evil to be assigned to the tectogrammatical categories $\text{Pr}_A$, $\text{Pr}_A \oplus \text{Pr}_N$, $\text{Pr}_A \oplus \text{Pr}_N \oplus \text{Pr}_P$, etc. Likewise, in a grammar with a theory of intonation, it would in prin-
ciple be possible to consider expressions which are NPs tectogrammatically, yet differ with respect to their phenotype – unaccented, accented, deaccented, etc. Grammar rules come in two varieties: **logical** rules, which tie together the behavior of various logical connectives, and **nonlogical** rules, which directly specify relationships between different classes of signs.

### 4.3.1 Categorical monads

Since the central point of this dissertation is not category-theoretic in nature, we do not wish to spend too much time doing category theory here. We present only a very brief introduction sufficient to discuss the idea of a categorical monad. The reader interested in pursuing category theory and the categorical underpinnings of logic is directed to [ML78], [Awo10], and [Gol14]. We assume some vague familiarity with the notion of categories, in particular the categorical notions of objects, morphisms (arrows), composition, and identity morphisms. When we write, for example, \( f: \mathcal{C} \to \mathcal{D} \), we read this as “\( f \) is a morphism whose domain is \( \mathcal{C} \) and whose codomain is \( \mathcal{D} \)”, or just “\( f \) is an arrow from \( \mathcal{C} \) to \( \mathcal{D} \)”. We will typically assume \( \mathcal{C} \) and \( \mathcal{D} \) are themselves categories. A morphism between categories is called a **functor**, and a morphism between any category \( \mathcal{C} \) and itself is called an **endofunctor**. We will often use \( \mathcal{F} \) and \( \mathcal{G} \) to denote functors. Of particular note is the identity endofunctor, written \( 1_{\mathcal{C}} \) for some category \( \mathcal{C} \). A morphism \( H: \mathcal{F} \to \mathcal{G} \) (for \( \mathcal{F} \) and \( \mathcal{G} \) functors) is called a **natural transformation**.

A **Monad** on a category \( \mathcal{C} \) is a triple \( < T, \eta, \mu > \) where \( T \) is an endofunctor of \( \mathcal{C} \), and \( \eta \) and \( \mu \) are natural transformations \( \eta: 1_{\mathcal{C}} \to T \) and \( \mu: T^2 \to T \) making the following diagrams commute, after [ML78]:

\[
\begin{array}{c}
T^3 \xrightarrow{T\mu} T^2 \\
\mu T \downarrow \quad \downarrow \mu \\
T^2 \xrightarrow{\mu} T
\end{array}
\]

This diagram can be thought of as expressing associativity, in that the order in which the \( \mu \) natural transformation composes with \( T \) is immaterial, i.e. that \( \mu \circ \mu T = \mu \circ T \mu \).
This diagram can be thought of as expressing two-sided identity, in that \( \mu \circ \eta T = \mu \circ T \eta = 1_C \).

We say an endofunctor on a monoidal category \( \langle C, \otimes, I \rangle \) with objects \( A \) and \( B \) (in this case, a monad) is **strong** if it has a natural transformation \( t_{A,B} : A \otimes TB \to T(A \otimes B) \).

We say a functor is **costrong** if it has a natural transformation \( t'_{A,B} : T(A \otimes B) \to A \otimes TB \). Each of these is subject to certain coherence conditions, and the interested reader is referred to [Koc72]. Costrength (of the cotensor corresponding to linear logic’s \( \to \)) will figure prominently when we get to the point of analyzing UCC.

### 4.3.2 Tectogrammar

While we will not adopt full classical linear logic, we will move towards a slightly more expressive system than the implicational fragment. We adopt the connective we refer to as additive disjunction (\( \oplus \), sometimes called “plus” and often interpreted as disjoint union in set-theoretic models), and add one which is not native to linear logic: the coordination monad operator \( \odot \), which can be thought of as a kind of modality. In [Tro92] the idea of a closure is introduced in an algebraic context, and our conception of monad is quite similar, albeit encoded in the deduction system itself. In the background for the tectogrammar we will additionally give rules for multiplicative conjunction (\( \otimes \), usually pronounced “tensor”), but our analysis does not hinge on this connective in any way; instead, it serves to show that \( \odot \) has the properties desired, namely that it is a strong monad. In this section we will first review the inference rules for the connectives of the tectogrammar, and then turn to a discussion of the additions we propose in order to account for unlike category coordination.

Up until now the notation of [Gir87] and [Tro92] does not differ: namely, we have used only one connective from LL, the linear implication \( \to \) pronounced “lolli”, “implies”, or “into”. However, once we add connectives, the lack of a uniform notation in the literature
becomes problematic. We primarily use the notation of [Gir87], which is perhaps more common, rather than that of [Tro92], which is perhaps more accessible to the linguistically inclined reader. We restate part of the table from chapter 2 so that the reader accustomed to [Tro92]’s notation can easily translate between the two as needed. The reader will note that this table is incomplete; we include only those connectives to which we will ultimately refer in the current chapter in one way or another, and their respective identities.

**Linear implication**

\[
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to I
\]

This rule encodes the fact that if a resource \( B \) is obtainable from a bag of resources \( \Gamma \) together with some resource \( A \to B \), that is, something “looking for an \( A \) in order to produce a \( B \)” is obtainable from \( \Gamma \) itself. It should be easily recognizable as abstraction, or as hypothetical proof.

\[
\frac{\Gamma \vdash A \to B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B} \to E
\]

This rule should be recognizable as a version of modus ponens, or as application. If we are able to produce a resource \( A \to B \) from some other resources \( \Gamma \) (potentially empty), and
we can produce $A$ from $\Delta$, then those two groups of resources can be combined in order to produce a resource $B$.

**Additive disjunction**

Linear logic already comes equipped with a connective that behaves in the desired manner, and since our tectogrammar is based on linear logic, we simply use that. We will need the additive disjunction type constructor $\oplus$, typically pronounced “plus”, whose associated inference rules are as follows:

$$
\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \text{⊕I}_1
$$

$$
\frac{\Gamma \vdash A}{\Gamma \vdash B \oplus A} \text{⊕I}_2
$$

These two introduction rules should be familiar to those acquainted with classical or intuitionistic logic, and should be thought of as encoding the fact that if a resource or expression is of some type $A$, then it will be in the disjunction of $A$ with another type $B$ (since $B$ is insignificant with respect to $A$), and this disjunction is commutative.\footnote{Actually, one introduction rule would be sufficient, with the other following as a theorem notably involving the rule of permutation or exchange.}

$$
\frac{\Gamma \vdash A \oplus B \quad \Delta, A \vdash C \quad \Delta, B \vdash C}{\Gamma, \Delta \vdash C} \text{⊕E}
$$

The elimination rule should likewise be familiar to those with a background in classical or intuitionistic logic. It says in effect that if the same resource $C$ can be obtained from either $A$ or $B$ (and some unknown resources $\Delta$), then that resource can also be obtained from whatever resources $\Gamma$ are suitable to produce their disjunction, that is, from a resource which is guaranteed to be either one or the other, though we may not know which ahead of time. This is sometimes referred to as “external choice”, that is, the choice of which resource is produced is not up to you.
The above rules target the tectogrammar only, and as such, we omit full grammar rules for them for the moment, noting that at least for the introduction rules, we will revisit this discussion in section 4.3.4. The case of the elimination rule is slightly more complicated, as we will see.

**Multiplicative conjunction**

The so-called “multiplicative conjunction” of linear logic is written $\otimes$ and usually pronounced “tensor”, or perhaps “times”.

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \otimes I$$

This rule says that two resources $A$ and $B$ may be combined into one composite resource. It is important to make careful note of the context here; we preserve all of the hypotheses used to produce both $A$ and $B$ in the conclusion.

$$\frac{\Gamma \vdash A \otimes B \quad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C} \otimes E$$

The elimination rule for $\otimes$ is in some ways reminiscent of the elimination rule for $\oplus$, in that the minor premiss contains hypotheses that are connected to the consequent of the major premiss, and the conclusion of the rule is identical. What this rule says in essence is that if you have some resource $C$ which is obtainable from resources $A$ and $B$, then a resource composed out of both of those formulas (that is, $A \otimes B$) would be sufficient to obtain $C$ as well.

**Additive conjunction**

As was the case with disjunction, LL provides a connective which will prove useful in analyzing constructions like those in (61): *Er findet und hilft Frauen*, the additive conjunction constructor $\&$, pronounced “with”. This connective is dual to the additive disjunction connective, and its rules reflect this duality. They will be familiar to anyone with a background
in classical or intuitionistic logic, as they are identical to the rules for the conjunctions from those systems. In linear logic, they can be thought of as encoding a kind of optionality, and are sometimes referred to as “internal choice”. From a single resource $A \& B$ it is possible to produce either an $A$ or a $B$, as you desire.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} \&I$$

This rule states that if you can produce from $\Gamma$ either an $A$ or a $B$, then you can also produce a single resource $A \& B$ which can be used as either one.

$$\frac{\Gamma \vdash A \& B}{\Gamma \vdash A} \&E_1$$

If you have a resource $A \& B$, then you can produce a resource $A$.

$$\frac{\Gamma \vdash A \& B}{\Gamma \vdash B} \&E_2$$

If you have a resource $A \& B$, then you can produce a resource $B$.

**Coordination monad**

We propose that the “best fit” for coordination, as far as logical connectives are concerned, is the unary modality $\Diamond$ from lax logic [FM97], which we refer to as laxity or colloquially “sometimes”. In category theoretic terms, this modality represents a strong monad, as noted in [Mog91], and in type theory, it is well known as encoding so-called computational types [BBdP98]. When we are in classical or intuitionistic logic, this modality is generally understood as meaning “true under some constraints”, but since linear logic is less concerned with truth, understanding it is somewhat less straightforward. In the case of coordination, a fundamentally non-linear process, the caveat of particular concern is “potential nonlinearity ahead”.

Rules for $\Diamond$:
The introduction rule for the coordination monad constructor \( \odot \) is quite simple. It says that any type \( A \) may also be treated as a coordinate structure of \( A \)'s. For the category theory-inclined reader, it should be obvious that this is the object-level part of the \( \eta \) natural transformation\(^8\) of a monad. For the reader more familiar with modal logic, this resembles the introduction rule for modal possibility. For the Haskell programmer, this construction is much like \texttt{return}. In the same way that we have canonical injections from, e.g., \( A \rightarrow A + B \) (or \( \oplus \), or \( \lor \), as may be your preference), we have a canonical injection from \( A \rightarrow \odot A \). So one might think of \( \odot \) being something like disjunction with an unknown formula, or moving from a more specific to a less specific state of information.

\[
\begin{array}{c}
\Gamma \vdash A \\
\hline
\Gamma \vdash \odot A
\end{array}
\]

At first glance, the elimination rule for \( \odot \) appears complicated, but again, categorical, logical, and computational perspectives can shed light on its function. This is perhaps most obvious with the case of Haskell’s \texttt{>>=} constructor (typically called “bind”), whose type, with \( m \) the ‘monad constructor’ is given as \( ma \rightarrow (a \rightarrow mb) \rightarrow mb \). In the inference rule, the major premiss represents the first argument \( ma \), and the minor premiss the second, \( (a \rightarrow mb) \), albeit in a form where the putative antecedent of the conditional is a hypothesis. The conclusion represents the return type \( mb \). In category theory, this resembles the \textbf{Kleisli star} operator \((\ast : (A \rightarrow TX) \rightarrow TA \rightarrow TX)\) with its arguments permuted.\(^9\)

From the logical perspective, we again find the disjunction metaphor to be useful, since we can see that the minor premiss relies on the formula under the scope of \( \odot \) in the major premiss. Since \( \odot \) can be thought of as representing the potential addition of unknown information, if we make use of the formula \( A \) in a (multiplicative) linear context, then we must

\[\eta : A \rightarrow TA\]

\(^8\)While this presentation differs somewhat from the categorical-style \( \eta / \mu \) presentation given earlier, it should be noted that the two presentations are intertranslatable.
‘transmit’ the fact that there is still potentially unknown information to be dealt with, and it is for this reason that the consequent of the minor premiss, and ultimately the conclusion of the rule, both appear under the scope of $\ominus$. It may be useful to compare this rule to a classical or intuitionistic disjunction ($\lor$) elimination rule. From a linguistic perspective, this rule encodes the fact that making use of an expression containing a coordinate structure results in an expression which also contains a coordinate structure.

Some lemmas about the coordination monad

For the sake of legibility, these are given in the tectogrammar only. Nevertheless, it is possible to give straightforward grammar rules demonstrating their applicability to the framework as a whole, and these rules follow in section 4.3.4. The exceptions are the lemmas involving $\otimes$, which exist here only to illustrate that the coordination monad connective $\ominus$ has the properties desired of a strong monad, and need not be used in any form by the grammar itself for the time being.

(45) Multiplication (M)

\[
\begin{align*}
\frac{\ominus \ominus A \vdash \ominus A}{\ominus A \vdash \ominus \ominus A} & \quad \frac{\ominus A \vdash \ominus A}{\ominus A \vdash \ominus A} & \quad \ominus E \\
\ominus \ominus A \vdash \ominus A & \quad \ominus A \vdash \ominus A & \quad \ominus I \\
\ominus \ominus A & \quad \ominus A & \quad \ominus \ominus A
\end{align*}
\]

This lemma should be recognizable to the category theorist as encoding the $\mu$ natural transformation\textsuperscript{10} of a monad. Linguistically speaking, this lemma says that a coordinate structure of coordinate structures is itself a coordinate structure. If we think of $\ominus$ as representing “unknown information”, then the lemma says that the addition of unknown information to a system which already contains unknown information results simply in a system with some unknown information. It is perhaps this insight that strikes most transparently at the nonlinearity of $\ominus$ in general, and of coordination in particular.

(46) Functoriality (F)

\[
\mu : T(TX) \rightarrow TX
\]

\textsuperscript{10}
This lemma shows that any functor $A \rightarrow B$ can be converted into a functor which takes as its argument an $A$ potentially containing a coordinate structure, and yields a $B$ potentially containing one as well. This remedies an oversight of [PH03]. Since this lemma relies on having an elimination rule for $\circ$, the omission of an elimination rule for the corresponding GEN[] constructor in that work to some extent forced an analysis whereby all functors were required to take coordinated arguments, necessitating a revision to the entire lexicon, and making it profoundly unclear what to do in the case of higher-order functors. We do not require that functors be forced to take coordinate-modal arguments lexically; instead their ability to do so comes for free as a theorem of the tectogrammar. Note that by replacing the context of the topmost major premiss (indicated in red) with $\Gamma$, i.e. $\Gamma \vdash A \rightarrow B$ instead of $A \rightarrow B \vdash A \rightarrow B$ and by stopping just short of the final introduction (indicated in blue), we obtain a version of the theorem in the manner of a derived rule:

$$
\frac{\Gamma \vdash A \rightarrow B}{\Gamma \vdash \circ A \rightarrow \circ B} \quad F
$$

(47) Cotensorial strength (CTS)

$$
\frac{A \rightarrow B \vdash A \rightarrow B}{A \rightarrow B, A \vdash B} \quad A \vdash A \rightarrow B
\frac{A \rightarrow B, A \vdash B}{A \rightarrow B, A \vdash \circ B} \quad \circ I
\frac{A, A \rightarrow B \vdash \circ B}{A \rightarrow B, A \vdash \circ B} \quad \circ E
\frac{\circ (A \rightarrow B) \vdash \circ (A \rightarrow B)}{\circ (A \rightarrow B), A \vdash \circ B} \quad \circ (A \rightarrow B) \vdash A \rightarrow \circ B
\frac{\circ (A \rightarrow B), A \vdash \circ B}{A, A \rightarrow B \vdash \circ B} \quad \circ E
\frac{\circ (A \rightarrow B) \vdash A \rightarrow \circ B}{\circ (A \rightarrow B), A \vdash \circ B} \quad \circ I
\frac{\circ (A \rightarrow B) \vdash A \rightarrow \circ B}{\circ (A \rightarrow B) \vdash A \rightarrow \circ B} \quad \circ I
\frac{\vdash \circ (A \rightarrow B) \rightarrow A \rightarrow \circ B}{\vdash \circ (A \rightarrow B) \rightarrow A \rightarrow \circ B} \quad \circ I
$$

In examining the endsequent of the cotensorial strength lemma, we can see that it is possible to produce a functor $A \rightarrow \circ B$ from $\circ (A \rightarrow B)$. This is useful in that it will allow for coordinated functors in the tectogrammar. In order to derive a coordinate functor
from two functors of type \( A \rightarrow B \), it becomes necessary to use \( \circ \)-introduction in order to produce objects of type \( \circ(A \rightarrow B) \). The reader will note, however, that such expressions are no longer themselves functors, and thus are unable to combine with their arguments via \( \rightarrow \circ \)-elimination, as would be the typical case. Fortunately, the cotensorial strength lemma shows that we can further derive the type \( A \rightarrow \circ B \) from \( \circ(A \rightarrow B) \), restoring the ability of the coordinate functor to combine with its arguments, albeit by “pushing” the modality down onto the return type. As was the case with the previous lemma, we can replace the context of the indicated premiss with \( \Gamma \) and stop short of the final introduction, yielding the following derived rule:

\[
\frac{\Gamma \vdash \circ(A \rightarrow B)}{\Gamma \vdash A \rightarrow \circ B} \quad \text{CTS}
\]

Under the metaphor whereby \( \circ \) indicates that a resource is available with certain caveats, then we can understand this lemma / rule to say, in essence: if we have a resource available that allows us to produce \( B \)s from \( A \)s (under conditions), then if we have an \( A \), we will be able to produce a \( B \), with those same extra conditions carried through. Linguistically speaking, as noted earlier, if we think of \( \circ \) as indicating “warning: potentially involves coordination”, then this is a perfectly sensible type to give the return type.

(48) Downward monotonicity of the first argument of \( \rightarrow \) (\( \rightarrow \)D1)

\[
\frac{\circ A \rightarrow B, \circ A \rightarrow B \quad \ fra{A \rightarrow A} \rightarrow_A \circ I \quad \ fra{A \rightarrow A} \rightarrow_A \circ I}{\circ A \rightarrow B, A \rightarrow B \rightarrow B} \rightarrow_I \circ B, A \rightarrow B \rightarrow B \rightarrow_I \circ B}
\]

This simply says that any functor that will accept a coordinated structure of \( A \)s as its argument will also accept a single \( A \) as its argument. For us, since \( \circ \) signifies the potential presence of a coordinate structure, it would be exceedingly odd to take a coordinated argument and return a type which is not likewise under the scope of the coordination monad constructor.
(49) Tensorial strength (⊗S)

\[
\begin{align*}
A \vdash A & \quad B \vdash B \\
\circ B \vdash \circ B & \quad A, B \vdash A \otimes B \\
\circ B, A \vdash A \otimes B & \quad \circ A, B \vdash (A \otimes B) \\
A \otimes B \vdash A \otimes B & \quad A, B \vdash (A \otimes B) \\
\circ B, A \vdash (A \otimes B) & \quad \circ A, B \vdash (A \otimes B) \\
\vdash (A \otimes B) \rightarrow (A \otimes B) & \quad \circ \circ I
\end{align*}
\]

This lemma shows that \( \circ \) forms a **strong** monad with respect to \( \otimes \).

(50) Tensorial costrength (⊗CS)

\[
\begin{align*}
A \vdash A & \quad B \vdash B \\
A, B \vdash A \otimes B & \quad \circ I \\
\circ A, B \vdash (A \otimes B) & \quad \circ P \\
\circ A \otimes B \vdash \circ A \otimes B & \quad \circ A, B \vdash (A \otimes B) \\
\circ A, B \vdash (A \otimes B) & \quad \circ \circ I
\end{align*}
\]

This lemma shows that the monad \( \circ \) is **costrong** as well. Taken together with \( \otimes S \), we have the immediate corollary that \( \vdash (\circ A \otimes B) \rightarrow (A \otimes B) \).

It is possible to give the derived rule of monadic application as well. Since we do not make specific use of this rule, we include the proof and the derived rule without further commentary:

(51) Monadic Application (∅App)

\[
\begin{align*}
A \rightarrow B \vdash A \rightarrow B & \quad A \vdash A \\
A \rightarrow B, A \vdash B & \quad \circ I \\
A, A \rightarrow B \vdash B \rightarrow \circ B & \quad \circ E
\end{align*}
\]

\[
\begin{align*}
\circ (A \rightarrow B) \vdash (A \rightarrow B) & \quad \circ A \vdash \circ A \\
A \rightarrow B, A \vdash \circ B & \quad \circ E \\
\circ (A \rightarrow B) \vdash (A \rightarrow B) & \quad \circ \circ I \\
\vdash (A \rightarrow B) \rightarrow (A \rightarrow B) & \quad \circ \circ I
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash (A \rightarrow B) & \quad \Delta \vdash \circ A \\
\Gamma, \Delta \vdash \circ B & \quad \circ \circ \circ \circ I
\end{align*}
\]

106
4.3.3 Phenogrammar

While the central point of chapter 3 was to extend LCG to LCG\(_\varphi\) by adding phenominators, such technology is essentially unnecessary here, since the phenotypes of the expressions with which we are concerned are simply St, and consequently they occur under the \(n\) phenominator, namely, the identity on strings. It will be much simpler to consider the grammar obtained by instantiating the coordination lexical entry from chapter 3 with strings as the phenotype, and by suppressing the phenominator entirely. This will eliminate upcasting and downcasting, an unnecessary complication for simple string conjunction.

Of course, this fails to generalize correctly with respect to functor coordination, in the manner described earlier. Nevertheless, it is entirely possible to recapitulate the analysis in this chapter by simply adding back the phenominators, and relevant upcasting and downcasting steps. We illustrate the necessary steps in how to analyze the tectogrammar of functor coordination with the coordination monad along the way anyway, since we will from time to time find it necessary to obtain a functorial category from a monadic category via the cotensor strength lemma.

It is not necessary to make any grand changes to the phenogrammar in order to analyze unlike category coordination. Since we have added new connectives to the tectogrammar, we will need to construct grammar rules for those connectives, but it is straightforward to construct rules in such a way as to maintain phenoterms from premisses to conclusions. The proposed lexical rules will present a slightly different problem, as some of them convert functor categories (attributive adjectives, for example) into simpler categories. In the case of the phenogrammar, we will be converting from string functions to strings. Fortunately we already have a function in place to perform this essential task: the function \texttt{say}, defined previously in chapter 3.
4.3.4 Grammar rules

The rules for \( \rightarrow \) in the tectogrammar correspond to the grammar rules of application and abstraction. The rules for \( \circ \) will correspond to the rules we will call Ret and Bind (after their Haskell counterparts). The rules for \( \oplus \) correspond to the rules Inj1 and Inj2, and for \&\&, we have Proj1 and Proj2. For the most part, the rules for \( \circ \) and \( \oplus \) leave the phenogrammar and the semantic component essentially unchanged. Since neither of these components themselves involve monadic types, we just make use of \( \lambda \)-abstraction and application in the usual manner, allowing the terms of the major premiss to substitute for a bound variable in the terms of the minor premiss.\(^{11}\) As before, in order to focus only on those issues which are germane to this dissertation, we provide only the phenogrammatical and tectogrammatical components of the rules. However, for the most part, the semantic components will be analogous to the phenogrammatical components.

Application and abstraction

As a reminder, we include the rules of application and abstraction here:

\[
\frac{\Gamma \vdash f : S \rightarrow T; A \rightarrow B \quad \Delta \vdash s : S; A}{\Gamma, \Delta \vdash (f \cdot s) : T; B} \quad \text{App}
\]

Since signs are triples in LCG, the grammar rule of application is just pointwise application of a functor sign to an argument sign, across all three components of each sign. If any component of the major premiss is not a functor whose antecedent type matches the type of the corresponding component of the minor premiss, then application may not proceed. When this situation arises, we refer to this as phenogrammatical, tectogrammatical, or semantic blocking, that is, blocking the successful derivation of the sign in question.

\[
\frac{\Gamma, x : S; A \vdash t : T; B}{\Gamma \vdash \lambda x : S. t : S \rightarrow T; A \circ B} \quad \text{Abs}
\]

\(^{11}\)More commonly this is done with so-called let-binding, but since we only use tectogrammatical monads here, there is no need to introduce new technology.
Similarly, the rule of abstraction proceeds pointwise in each component, over the occurrence of a hypothetical sign in the context. The implications created are those of each of the component logics, with linear implication in the tectogrammar, and the implication of higher order logic in the phenogrammar.

Additive disjunction

\[
\frac{\Gamma \vdash s : S; A}{\Gamma \vdash s : S; A \oplus B} \text{ Inj}^1
\]

\[
\frac{\Gamma \vdash s : S; B}{\Gamma \vdash s : S; A \oplus B} \text{ Inj}^2
\]

Since the disjunction we are interested in making use of lives at the level of the tectotypes, we do not see a need to complexify the pheno or the semantics with respect to \( \oplus \).

To that end, we simply maintain the pheno and semantic judgments entire from premiss to conclusion, and modify only the tectotype. These rules say, for a sign given a tectotype \( A \) (respectively, \( B \)), we may produce a sign of tectotype \( A \oplus B \), with no change to the phenogrammatical component.

Writing an elimination rule is less straightforward, as it would seem to need to involve \texttt{case ... of} terms in the pheno and semantics on the grounds that variable choice in the hypotheses of the minor premisses must be fresh. As observed by Morrill in [Mor94], English (and perhaps natural language in general) does not seem to motivate the elimination rule for \( \oplus \). In absence of evidence to the contrary, we agree, and remain unconcerned with a precise formulation of this rule merely for the sake of completeness. Additionally, since we do not allow for the formation of disjoint unions in the other two components, and since we do not use tectoterms, there is no reason to define \texttt{case ... of} term constructors, although adding them to the system is relatively straightforward if desired.\(^{12}\)

\(^{12}\)Pollard (p.c.) suggests that this provides evidence that disjunctive types are the wrong way to go, and that we would be better served with subtyping and inclusion postulates along the lines of [Pol14], an idea borrowed from pregroup grammar [Lam99]. Nevertheless, since we will eventually need \( \oplus \) for independent reasons (multiset types), we feel that the connective may as well be pressed into service here as well.
Additive conjunction

\[
\frac{\Gamma \vdash s : S; A & B}{\Gamma \vdash s : S; A} \text{ Proj1}
\]

\[
\frac{\Gamma \vdash s : S; A & B}{\Gamma \vdash s : S; B} \text{ Proj2}
\]

The reader will note the strong parallelism with the additive disjunction rules given immediately prior. We do not generally take the choice of one of the additive conjuncts to have phenogrammatical or semantic reflexes. In a similar but contrasting case to additive disjunction, English does not appear to motivate introduction rules for additive conjunction, a fact which is also noted in [Mor94]. In this dissertation we make use of \& only for the sake of theoretical completeness, sketching an analysis of a famous example of argument neutrality from German as reported in [PZ86].

Coordination monad

\[
\frac{\Gamma \vdash s : S; A}{\Gamma \vdash s : S; \odot A} \text{ Ret}
\]

The Ret rule (mnemonic for return, a locution borrowed from Haskell) can be thought of as a sort of unary disjunction rule. The reader may note the similarity between Ret and the rules Inj1 and Inj2, which underscores the point. Essentially this rule says that any expression of tectotype A may be treated as though it might involve a coordinate structure, and so is justified in being given the type \(\odot A\). The phenogrammatical component, as was the case with Inj1 and Inj2, remains unchanged from premiss to conclusion.

\[
\frac{\Gamma \vdash s : S; \odot A}{\Gamma, \Delta \vdash (\lambda x : S. t) s : T; \odot B} \text{ Bind}
\]

The rule Bind (again, borrowing suggestive Haskell nomenclature) is perhaps the least intuitive. Focusing first on the tectogrammatical component, we can see that the rule is identical to the rule of \(\odot\)-elimination from the discussion in section 4.3.2; the major
premiss bears a coordination monad type in its consequent, and the minor premiss contains a hypothesis of the same type, without the coordination monad constructor. The consequent of the minor premiss again is under the scope of the coordination monad constructor. In the conclusion of the rule, the new sign has a coordinate structure tectotype as well.

4.4 Analysis

4.4.1 Lexical entries

Since this section replicates much of section 3.3.2, we will only repeat and comment on those lexical entries about which more should be said by virtue of their participation in unlike category coordination. In addition, we give lexical entries for a couple of new verbs which take predicative complements, and revise the lexical entry for the coordination conjunction and. Everything else may be safely assumed to be the same as it has been up to now.

Adjectives

(52) \( \vdash (\text{r evil}) : \text{St}_1; \text{N} \rightarrow \text{N} \)

As noted previously, adjectives map common nouns to common nouns (type \( \text{N} \rightarrow \text{N} \)) in the tectogrammar. In the phenogrammar, they are unary string functions under the image of the \( \text{r} \) phenominator, indicating that they linearize their argument to the immediate right of their string support, in this case the string constant evil.

It would be possible to construct a nonlogical rule mapping between adjective types and predicative adjective types, but such a rule is poorly motivated. We consider first the case of predicative adjectives requiring (or allowing) complements, as in:

(53)  
a. Tyrion was [fond of wine].  

Here, *fond of wine* is acting as an AP in the predicative case, yet it cannot function as a nominal premodifier as indicated. So we will be unable in general to map from predicatives to modifiers. The same example serves to illustrate why we cannot have a mapping in the other direction as well: on such a story, in order for *fond of wine* to appear in a predicative position, it must have originated as a modifier. Clearly, this is not possible either. Further motivation is provided by adjectives such as *alone*, which appear only in a predicative position. So we are left to conclude that for the time being, the most perspicuous thing to do is simply to list predicative adjectives lexically, in the following manner:

(54) ← evil : St; PrA

**Prepositions**

(55) (t on) : St2; NP ← N → N

As noted earlier, Locative prepositions take noun phrase arguments in order to produce common noun modifiers of type N → N, the same type as adjectives. Phenogrammatically they are under the “transitive” phenominator t, indicating that they linearize their first argument to the immediate right of their string support *on*, and their second argument to its immediate left.

Complications arise due to the fact that certain prepositions known as “case-marking” prepositions appear in verbal complement PPs:

(56) Stannis relied [on Melisandre].

The distribution of these is known to be idiosyncratic and necessitates the addition of new types, so when it comes to predicative prepositional phrases, we will establish a mapping between examples like (55) and the predicative category.
Verbs of predication

Here, we provide lexical entries for the copula and the verb become, based on their selectional properties as illustrated in section 4.1.1. Since the disjunctive tectotypes at times become cumbersome, we make the following abbreviations:

\[
\text{Prd} = \text{def} \ PrA \ominus \text{PrN} \ominus \text{PrP} \\
\text{AoN} = \text{def} \ PrA \ominus \text{PrN}
\]

Now we can provide a simplified lexical entry for the copula. As noted previously, we are not concerned with copular agreement paradigms here, and we make the further simplifying assumption that both the copula and become select only for NP-type subjects\(^{\ref{footnote:13}}\), as opposed to \(\overline{S}\), PP-subjects, dummy it, and the like. Extending the formalism to cover this kind of subject transparency can be done straightforwardly by treating these verbs as raising verbs of tectotype \( (A \rightarrow \text{Prd}) \rightarrow (A \rightarrow S) \) (respectively \( (A \rightarrow \text{AoN}) \rightarrow (A \rightarrow S) \)), where \( A \) ranges over the relevant collection of subject categories.

\[
(57) \vdash (t \text{ is}) : \text{St}_2; \text{Prd} \rightarrow \text{VP}
\]

The lexical entry for the copula indicates that it is phenogrammatically transitive, i.e., its string support is occurs under the t phenominator, and that it selects for a predicative complement in order to produce a verb phrase. For the verb became, we provide a similar lexical entry to our entry for the copula, noting that it differs only in its tectotype, where the first argument is either a predicative adjective or a predicative NP as desired:

\[
(58) \vdash (t \text{ became}) : \text{St}_2; \text{AoN} \rightarrow \text{VP}
\]

Conjunctions

Now we can give a revised lexical entry schema for and, extending the one appearing in (34) with the coordination monad type constructor:

---

\(^{\ref{footnote:13}}\)The system described here allows for coordinated NP subjects as well, although as previously noted, it overgenerates with respect to the copula.
\[
(59) \quad \begin{aligned}
\lambda \psi \cdot (\lambda P \cdot P \cdot (\text{say}_{P \psi} c_1) \cdot \text{AND} \cdot (\text{say}_{P \psi} c_2)) : \\
&P \psi \to P \psi \to P \psi; \\
\circ A \rightarrow \circ A \rightarrow \circ A
\end{aligned}
\]

The phenogrammar of the sign is unchanged from the previous lexical entry: it takes two arguments under the image of some phenominator \( \psi \) (that is, strings, or functions over strings having a contiguous string support) and produces an expression of the same type. This is achieved by independently applying the \text{say} function to each conjunct, concatenating them with the string constant \text{AND} in between, and then applying the phenominator \( \psi \) to the resulting string. The sole locus of change is to the tectotype of the sign. Whereas we had previously given it the type schema \( A \rightarrow A \rightarrow A \), this will not suffice, so in order to ensure that UCC may proceed in a sensible fashion, we give the conjunction \text{and} the tectotype \( \circ A \rightarrow \circ A \rightarrow \circ A \). Now we have a sign that tectogrammatically takes as input two expressions (potentially involving coordination), and returns one which involves coordination. We thus ensure that simple types can be coordinated, as well as disjoint union types, and coordinate structures themselves, and that none of these typing judgments suffer from the issues discussed in section 4.2.1. In theory, it would be possible to let \( \circ \) be the identity monad, but this does no good, as defining \( \circ \) (for some type \( A \)) to be \( 1_A \) places us back in the same origin category, when the entire point is to wind up in a slightly larger category.

### 4.4.2 Nonlogical rules

In addition to \textbf{logical} rules, which correspond to inference rules governing the behavior of logical connectives, LCG has \textbf{nonlogical} rules, which allow for direct mapping from one particular kind of sign to another. These kinds of rules are entirely equivalent to having a lexical entry whose phenogrammatical component is just the identity on its input type, but we choose to encode this process in the rule system, on what amount to aesthetic grounds.
With the case of the predicatives, nonlogical rules simplify the picture considerably. While verbs embedding predicatives do take disjunctive arguments, that is, they are neutral with respect to the category of expression they embed in particular ways, there is some benefit to singling out predicative categories as being different than the source categories we are mapping from. By way of example noted previously, it is not necessarily the case that all predicative adjectives are attributive, so having \textbf{PrA} as a separate tectogrammatical category is well-motivated, and it is to this category that we would lexically assign expressions like \textit{alone}.

However, mapping between tectotypes is not enough. Since these are grammar rules, albeit of a different kind, they are rules governing the manipulation of LCG signs, and so they must address the phenogrammatical and semantic components as well. Fortunately, we are confident in giving a uniform phenogrammatical representation to predicatives: we assign them the category St by using \texttt{say}, as discussed in section 3.2.3.

\[
\frac{\Gamma \vdash p : \text{St}; \text{NP}; \text{PNP}}{\Gamma \vdash p : \text{St}; \text{PrN}; \text{PNP}}
\]

The rule of predicate noun phrase formation (PNP) takes as input a sign which is, tectogrammatically speaking, a simple (i.e. non-quantificational) noun phrase, and returns a predicative NP category (PrN). Phenogrammatically, it simply carries the string \(p\) through intact in the output of the rule.

\[
\frac{\Gamma \vdash p : (\text{St\,-\,2})_q; \text{VP} \rightarrow \text{S}; \text{PQNP}}{\Gamma \vdash \texttt{say}_{(\text{St\,-\,2})_q} p : \text{St}; \text{PrN}; \text{PQNP}}
\]

The rule of predicate quantified noun phrase formation (PQNP) is slightly more complicated. In the phenogrammar, we again apply \texttt{say} to an argument under the image of the QNP phenominator. In this case, \texttt{say} passes in the identity function on strings, guaranteeing that we are returned a representation of the expression’s string support. In the tectogrammar, we map the type of QNPs \(((\text{NP} \rightarrow \text{S}) \rightarrow \text{S})\) to the predicative type PrN. Strictly speaking, this rule is unnecessary, as one could simply hypothesize a noun phrase,
conjoin, then withdraw the hypothesis, and lower the QNP into place in a manner similar to the one illustrated in (29). Here, we provide this rule purely as a derivational convenience: that is, it will be possible to create predicate nominals directly from QNPs, rather than adding back in the extra hypothetical and abstraction steps.

$$\Gamma \vdash p : (St_1); N \rightarrow N; \quad \Gamma \vdash \text{say}_{(St_1)}; p : St; PrP; \quad \text{PPP}$$

Predicative prepositional phrases are handled much the same way as predicative adjectives with one notable exception: the phenominator of the input expression in the phenogrammar differs. So the rule of predicate prepositional phrase formation (PPP) takes as input a sign which, first, is phenogrammatically a function from strings to strings, under what we have called the VP phenominator $i$ (i.e. one looking leftward for its argument), and returns a string by applying the function $\text{say}$. In the tectogrammar, we map between a common noun modifier type ($N \rightarrow N$) to a predicative PP type ($PrP$).

### 4.4.3 Example proof

1. $(\det \ a) : St \rightarrow St_1 \rightarrow St; N \rightarrow VP \rightarrow S$

   This is the lexical entry for the indefinite determiner.

2. $\vdash \text{TYRANT} : St; N$

   Here we have the lexical entry for the common noun $\text{tyrant}$.  

116
3. \( \vdash (q \cdot \text{TYRANT}) : \text{St}_{-2}; \text{VP} \rightarrow S \)

We apply \( a \) to \text{tyrant} to construct the QNP \( a \text{ tyrant} \).

4. \( \vdash P_a (q \cdot \text{TYRANT}) \)

The subtyping predicate holds of the term \((q \cdot \text{TYRANT})\) for the phenominator \( q \).

5. \( \vdash \downarrow_q (q \cdot \text{TYRANT}) : (\text{St}_{-2})_q; \text{VP} \rightarrow S; \)

Since the subtyping predicate was fulfilled, we can downcast into the relevant subtype.

6. \( \vdash \text{say}_{(\text{St}_{-2})_q} \downarrow_q (q \cdot \text{TYRANT}) \)

\begin{align*}
&= \text{say}_{\text{St}_{-2}} (q \cdot \text{TYRANT}) \\
&= \text{say}_{\text{St}_{-2}} (\lambda P. P (\cdot \text{TYRANT})) \\
&= \text{say}_{\text{St}} (\lambda P. P (\cdot \text{TYRANT}) \text{ vac}_{\text{St}_{-1}}) \\
&= \text{say}_{\text{St}} (\lambda P. P (\cdot \text{TYRANT}) \lambda s.s) \\
&= \text{say}_{\text{St}} (\lambda s.s (\cdot \text{TYRANT})) \\
&= \lambda s.s (\cdot \text{TYRANT}) \\
&= a \cdot \text{TYRANT} : \text{St}; \text{PrN} \\
\end{align*}

The predicative QNP formation rule specifies that we apply the pheno of a QNP to the identity on strings.

7. \( \vdash a \cdot \text{TYRANT} : \text{St}; \text{AoN} \)

We inject the predicate nominal type PrdN into the sum type AoN, that is, \( \text{PrA} \oplus \text{PrN} \).

8. \( \vdash a \cdot \text{TYRANT} : \text{St}; \circ \text{AoN} \)

We inject the sum type into the unary coordinate structure type using the \textbf{Ret} rule.

9. \( \vdash P_a a \cdot \text{TYRANT} \)

The subtyping predicate holds of the string \( a \cdot \text{TYRANT} \) for the phenominator \( n \).

10. \( \vdash \downarrow_n a \cdot \text{TYRANT} : \text{St}_n; \circ \text{AoN} \)

Since the subtyping predicate was fulfilled, we can downcast into the relevant subtype.
11. \( \vdash \lambda c_1 : \text{St}_n. \downarrow_n (n ((\text{say}_{\text{St}_n} c_1) \cdot \text{AND} \cdot (\text{say}_{\text{St}_n} c_2))) \)
\( = \lambda c_1 : \text{St}_n. \downarrow_n (n ((\text{say}_{\text{St}_n} (\uparrow_n c_1)) \cdot \text{AND} \cdot (\text{say}_{\text{St}_n} (\uparrow_n c_2)))) \)
\( = \lambda c_1 : \text{St}_n. \downarrow_n (n ((\lambda s.s (\uparrow_n c_1)) \cdot \text{AND} \cdot (\lambda s.s (\uparrow_n c_2)))) \)
\( = \lambda c_1 : \text{St}_n. \downarrow_n (n ((\uparrow_n c_1) \cdot \text{AND} \cdot (\uparrow_n c_2))) \)
\( = \lambda c_1 : \text{St}_n. \downarrow_n (\lambda s.s ((\uparrow_n c_1) \cdot \text{AND} \cdot (\uparrow_n c_2))) \)
\( = \lambda c_1 : \text{St}_n. \downarrow_n (((\uparrow_n c_1) \cdot \text{AND} \cdot (\uparrow_n c_2)) : \text{St}_n \to \text{St}_n; \circ \text{AoN} \to \circ \text{AoN} \to \circ \text{AoN} \)

We instantiate the lexical entry for \textit{and} with \( A \) as type \text{St}, \( \varphi \) as the phenominator \( n \), and tectotype \text{AoN}, yielding the by-now-familiar phenoterm for straightforward string coordination, and with the the coordinate-modal sum type \( \circ \text{AoN} \) in the tectogrammar. As a reminder, in order to save space in future derivations we will customarily omit the entire coordination lexical entry, leaving instead just the string constant \textit{AND} as a mnemonic for the reader.

12. \( \vdash \lambda c_1 : \text{St}_n. \downarrow_n ((\uparrow_n c_1) \cdot \text{AND} \cdot \textit{TYRANT}) : \text{St}_n \to \text{St}_n; \circ \text{AoN} \to \circ \text{AoN} \)

The expression \textit{a tyrant} becomes the second conjunct.

13. \( \vdash \text{evil} : \text{St}; \text{PrA} \)

Here we have the original lexical entry for \textit{evil}.

14. \( \vdash \text{evil} : \text{St}; \text{AoN} \)

We use the rule \textbf{Inj1} to ‘weaken’ the type by injecting the source type \text{PrA} into the sum type \text{PrA} \( \oplus \text{PrN} \), represented here as \text{AoN}.

15. \( \vdash \text{evil} : \text{St}; \circ \text{AoN} \)

We inject the resulting expression into the unary coordinate structure containing that expression by introducing the monad operator \( \circ \) via the \textbf{Ret} rule.

16. \( \vdash P_n \text{evil} \)

The subtyping predicate holds of the term \textit{evil} for the phenominator \( n \).
17. \( \vdash \downarrow_{\text{evil}} \colon \text{St}_n; \odot \text{AoN} \)

Since the subtyping predicate was fulfilled, we can downcast into the relevant subtype.

18. \( \vdash \downarrow_{\text{evil} \cdot \text{and} \cdot \text{a} \cdot \text{tyrant}} \colon \text{St}_n; \odot \text{AoN} \)

Finally we can apply the partial coordinate structure to construct the entire coordinate structure \textit{evil and a tyrant}. The coordinate structure still bears its phenogrammatical subtyping through the downcasting operator \( \downarrow_n \), and the tectotype \( \odot \text{AoN} \), that is, \( \odot(\text{PrA} \oplus \text{PrN}) \).

19. \( \vdash \text{evil} \cdot \text{and} \cdot \text{a} \cdot \text{tyrant} \colon \text{St}; \odot \text{AoN} \)

We upcast the coordinate structure into its phenogrammatical supertype \( \text{St} \).

20. \( \vdash (\text{t became}) : \text{St}_2; \odot \text{AoN} \rightarrow \text{VP} \)

This is the lexical entry for the verb \textit{became}.

21. \( \vdash (\text{t became}) : \text{St}_2; \odot \text{AoN} \rightarrow \odot \text{VP} \)

We use the functoriality lemma to ensure that the verb \textit{became} can take a coordinate structure as its first argument.

22. \( \vdash (\text{i became} \cdot \text{evil} \cdot \text{and} \cdot \text{a} \cdot \text{tyrant}) : \text{St}_1; \odot \text{VP} \)

We apply \textit{became} to the newly-formed coordinate structure to create a (coordinate-modal) VP.

23. \( \vdash (\text{i became} \cdot \text{evil} \cdot \text{and} \cdot \text{a} \cdot \text{tyrant}) : \text{St}_1; \text{NP} \rightarrow \odot \text{S} \)

We use the cotensorial strength lemma to ensure that verb phrase \textit{became evil and a tyrant} can take a non-coordinate noun phrase as its subject argument.

24. \( \vdash \text{joffrey} : \text{St}; \text{NP} \)

This is the lexical entry for the name \textit{Joffrey}.

25. \( \vdash \text{joffrey} \cdot \text{became} \cdot \text{evil} \cdot \text{and} \cdot \text{a} \cdot \text{tyrant} : \text{St}; \odot \text{S} \)

To finish, we apply the VP to its subject, and get the expression \textit{Joffrey became}
evil and a tyrant, which, as desired, is a string in the phenogrammar and a sentence (involving coordination) in the tectogrammar.

4.4.4 Argument neutrality

Another kind of unlike category coordination occurs with so-called argument neutrality, where two functors with different argument selection properties appear in a coordinate structure. While such examples are rare in English, these kinds of structures can be analyzed in LCG with the addition of the additive conjunction (&, pronounced “with”) from linear logic, which is essentially similar to the more familiar boolean conjunction operator ∧. We have in mind examples from German such as the following from [PZ86], repeated here with our numbering, though the original examples are (37) and (40) in the text:

\[(60)\]
\[
\begin{align*}
\text{a. } & \text{* Sie findet und hilft Männer.} \\
& \text{she finds and helps men.ACC.PL}
\end{align*}
\]
\[
\begin{align*}
\text{b. } & \text{* Sie findet und hilft Männern.} \\
& \text{she finds and helps men.DAT.PL}
\end{align*}
\]

\[(61)\]
\[
\begin{align*}
\text{Er findet und hilft Frauen.} \\
& \text{he finds and helps women.ACC/DAT.PL}
\end{align*}
\]

The crucial observations here are that the verb *finden* (inflected here as *findet*) selects an accusative object, and *helfen* (likewise *hilft*) a dative object. When the objects of the coordinated VP are morphologically accusative or dative, as in (60a) and (60b) respectively, the resulting sentence is ungrammatical. However, since the noun *Frauen* is syncretic between accusative and dative, the coordination proceeds impeccably in (61).

Our analysis here is in essence an LCG recoding of the one given in [PH03], which is itself theoretically indebted to [Bay96] and to [Mor94]. Owing to the fact that German clause structure is somewhat more complex than English, we provide lexical entries sufficient to analyze the verb phrases of such sentences (omitting by necessity virtually all other details about German grammar).
The preceding noun phrases are also strings in the phenogrammar. We give the NP Männer the tectotype Acc, for accusative noun phrases, and we give its dative counterpart Männern the tectotype Dat. The noun phrase Frauen is syncretic between the two cases, and so we give it the tectotype Acc & Dat, which will allow it to function tectogrammatically as either type Acc or type Dat, as desired, by virtue of the rules Proj1 and Proj2.

The two verbs in question are not very much different from English transitive verbs. They both appear as binary string functions under the image of the t phenominator, and they both form verb phrases (the type GVP, for “German VP”) when they combine with their objects. The only difference between the two is that findet selects for an object argument of type Acc, and hilft for a dative object (type Dat).

Finally, we have the coordinating conjunction und, to which we give a lexical entry schema virtually identical to the one for English given in (59). We now have enough technology to be able to analyze the construction in question, and a full proof is given in section 4.6. We add the following abbreviation in order to save space: AwD = def Nom & Dat.

As for blocking the ungrammatical sentences, it suffices to note that a proof along similar lines will not go through, since while it is possible to obtain generally a syntactic category \((A & B) \rightarrow C\) from either \(A \rightarrow C\) or \(B \rightarrow C\), there is no way to obtain \(A \rightarrow C\) from \(B \rightarrow C\),
or vice versa. So each verb is unable to select for the complement of the other on their own. Likewise, it is not possible to obtain a type \((A \& B)\) from either type \(A\) or type \(B\), so we will be unable to create an expression capable of being the object of the coordinated verb phrase in the case of either the purely accusative \(Männer\) or the purely dative \(Männern\), which is exactly what we want.

Much like German, English case syncretism can be modeled using \& as well, so its inclusion as a connective is independently motivated. This enables a distinction between the nominative pronouns \(I/he/she/we/they\) and the accusative pronouns \(me/him/her/us/them\), and allows proper names like \(Tyrion\) to appear in the positions where either can appear.

\subsection*{4.5 Discussion}

While predicatives and unlike category coordination have been reasonably well-studied, most accounts suffer from some variety or other of problem relating to the category assigned to the resulting coordinate structure. Whitman [Whi02] and Pollard and Hana [PH03] are notable exceptions, though Whitman’s solution perhaps incorrectly associates coordination with continuations, and Pollard and Hana’s fails to generalize to cases of functor coordination, and impels unnecessary lexical revision.

We have shown how such constructions can be analyzed in LCG through the addition of two tectogrammatical connectives: the additive disjunction connective \(⊕\), and the coordination monad \(◯\). Several nonlogical rules are motivated in order to map from categories which are logically derived, and their predicative counterparts.

A number of useful theorems can be proven regarding \(◯\), and these in turn will pay off when it comes to providing a proof-theoretic analysis of UCC. The inference rules of \(◯\) necessitate some minor additions to the term calculi of the LCG phenogrammatical and semantic components. We have shown that the correct strings can be derived, given accurate tectogrammatical category assignments, and sketched a semantics using generalized
conjunction and predicate formation functions.

Finally, we provide a preliminary analysis of asymmetric coordination within relative clauses, and give a number of proofs sufficient to analyze the examples in the text.

4.6 Additional proofs

Here we provide additional proofs deriving the example sentences given earlier.
Figure 4.1: Derivation for Robb and Catelyn died
Figure 4.2: Derivation for *Joffrey whined and sniveled*
Figure 4.3: Derivation for *Joffrey is evil*
Figure 4.4: Derivation for Joffrey is Aerys the third
Figure 4.5: Derivation for *Joffrey is a tyrant*
Figure 4.6: Derivation for Joffrey is on the iron throne
Figure 4.7: Derivation for \textit{Joffrey is a coward and a tyrant}
Figure 4.8: Derivation for *Joffrey is evil and a tyrant*
Figure 4.9: Derivation for *Joffrey is on the iron throne and a tyrant*
Figure 4.10: Derivation for *Joffrey is evil and on the iron throne*
Figure 4.11: Derivation for Joffrey is evil and a tyrant and on the iron throne
Figure 4.12: Derivation for *findet und hilft Frauen*
5.1 Overview and data

It has been suggested previously, notably in [Moo15], that curryesque grammars that feature linear logic as their centerpiece are ill-equipped to handle coordination, both for reasons of directionality, and because coordination is an essentially nonlinear phenomenon. LCG can address the first criticism by adding phenominators, and the second by adding monad tectotypes. However, there is another way in which coordination exemplifies nonlinearity. **Iterated coordination** is the phenomenon whereby the initial conjunct of a coordinate structure may itself be composed of more than one expression. These do not themselves necessarily appear as coordinate structures, but instead as lists (though lists of coordinate structures are certainly possible). In general, these lists appear to follow the same general rules about coordinability as do other coordinate structures. Strings can be coordinated:

(11) Tyrion hated [[Joffrey, Tywin,] and [Cersei]]

Here we refer to the list containing the noun phrases as the **iterated conjunct**. The entire coordinate structure itself is exemplary of iterated coordination. The iterated conjunct can contain arbitrarily many (potential) conjuncts, each of which is syntactically interchangeable. Semantically and pragmatically speaking, such lists appear to be sensitive to both temporal and dynamic effects, as is generally the case for coordination:

(12) Olenna [[hated, poisoned,] and [killed]] Joffrey.

(12') # Olenna [[killed, hated,] and [poisoned]] Joffrey.
Tywin chastised [[his daughter], [her son]], and [[the son]’s wife].

The previous example shows additionally that functors may also occur in iterated constructions. Since functors in (most) curryesque grammars are represented as functions in the phenogrammar, curryesque CGs wishing to analyze iterated coordination must allow not only functor coordination but functor iteration. This is similarly challenging to non-iterated coordination in that ultimately the strings corresponding to the string support of the functors appear concatenated, yet the ability to combine with other expressions must in some way be restored to the entire coordinate functor. Fortunately, the same mechanism by which functor coordination was enabled for LCGϕ can be marginally extended to cover such cases.

Unlike category coordination participates in iterated coordinate structures:

(13) Joffrey became [[evil, a tyrant,] and [deceased]].

Our analysis of UCC will remain unchanged, and we will see that the additions proposed here will be able to successfully analyze these cases as well.

Lists are motivated for English independently of explicit coordination, as in the following construction:

(66) Q: Who is on Arya’s list?

A: Joffrey, Cersei, Walder Frey, Meryn Trant, Tywin Lannister, ...

Furthermore, it has been observed (in [Zim00] and implicitly in [Pru08], for example) that such lists typically have characteristic prosody associated with them, independent of coordinate structures they may appear in, so it is not unreasonable to assume that the list forms a structure in and of itself, and it is an analysis along these lines that we will pursue here.
5.1.1 Analytical strategy

It is fairly straightforward to extend LCG with types and constructors sufficient to analyze
iterated coordination. Phenogrammatically speaking, we will want to be able to form lists
of expressions, and tectogrammatically we will form multisets. We will need to revise our
lexical entry schema for coordination in order to account for iterated conjuncts. This will
necessitate additional clauses to the definition of the function \texttt{say}, though phenominators
will remain essentially unchanged.

Following this overview, we discuss in section 5.2 how to add list and multiset types,
which we refer to collectively as \textit{iterated types}, to LCG in order to provide an analysis
of iterated coordination. We discuss inter alia the construction of \texttt{cons} lists as phenoterm
constructors. We revise the definition of the \texttt{say} function originating in chapter 3. We
provide grammar rules for manipulating expressions involving the new types. In section
5.3, we propose an analysis for iterated coordination based on modifying the lexical entry
for \texttt{and} in order to make reference to list and multiset types. We show that the relevant
examples are derivable in this system. Finally, section 5.4 summarizes these results and
presents some issues for future research.

5.1.2 Comparison with previous approaches

Regrettably little work has been done on this topic in the categorial grammar literature. As
far as we know, the first categorial grammar treatment of iterated coordination is evidently
from [Mor90]. In [Mor94], he briefly considers an analysis using \textbackslash{\textmu}, which he describes as
bearing the same relationship to his other connective \textmu as does linear logic’s “why not”
exponential \textmu to its counterpart “of course” (!). The primary difference is that \textmu and \textbackslash{\textmu}
allow for contraction and respectively, expansion, but not weakening. As is the case with
the tectogrammatical component of our analysis, the one from [Mor94] forms multisets of
formulas in the type logic, and these are interpreted into string-based phonologies.

\footnote{A volume which we have at the time of writing unfortunately been unable to obtain.}
While the motivation for iterated types and analysis thereof in [Sch05] is superficially similar in the use of the Kleene star operation, he argues that an analysis of iterated coordination must invoke concatenation as a primitive instead of cons lists, because of the perceived necessity via data from [Hud89] of distributing a functor over arbitrary sublists of iterated lists, not necessarily including the coordinating conjunction. It so happens that our judgments are not in line with the data given in both [Sch05] and [Hud89], and we suggest that such examples are accommodation. In particular, the “in two, but not the other one” reading given for [Sch05]’s (13b), repeated here with our numbering, does not seem correct:

(67) either in England, in the United States or (the) Netherlands

Regardless, this is an empirical question, and one about which we must reserve judgment.

Both [Mor94] and [Sch05] express their analyses in some version or other of extended Lambek grammar, and as such, their analyses are not directly available to LCG given its curryesque nature. Furthermore, both make use of concatenation over cons lists. Since our analysis of coordination requires the ability to define say with respect to any number of different types, the choice to use cons lists as representations for the terms inhabiting iterated types is one which is technically rather than empirically motivated. Nevertheless, since linear logic’s ? and the Kleene star are both known to be monads, our analysis will bear some similarity to that of both [Mor94] and [Sch05], albeit couched in terms of a curryesque grammar formalism. Unlike our analysis of unlike category coordination, in order to analyze iterated coordination, we will need to invoke different monads in each component of the grammar.

As was the case with unlike category coordination, our analysis is more in the spirit of [PH03]. One area in which we depart from them is the fact that our notion of tectogrammar is based on an elaboration of linear logic, rather than higher order logic. Due to the unordered nature of linear logic (in particular, due to the fact that its multiplicative product ⊗ is commutative), it is not immediately clear what it would even mean to have
list types in the tectogrammar\textsuperscript{2}. We will instead make use of lists in the phenogrammar, and finite multisets in the tectogrammar. As noted previously, the analysis in [PH03] is not sufficient to analyze functor coordination, given its apparent inability to distinguish between “rightward-looking” and “leftward-looking” functors, for example. This is obviated in LCG\textsubscript{$\varphi$} as shown in chapter 3.

5.2 Adding iterated types to LCG

In order to provide a formal analysis of iterated coordination, we will need to find a way to collections of coordinable expressions. In the tectogrammar, these will take the form of types representing finite multisets (sometimes called bags), i.e. types whose inhabitants are unordered collections of resources. These will be defined in terms of the multiplicative conjunction $\otimes$ from linear logic, together with a slight generalization of the additive conjunction $\oplus$ to countable coproducts. In the phenogrammar, these objects will correspond to lists of expressions.\textsuperscript{3} While we will omit semantics here, it is reasonable to assume that, for reasons of, say, anaphora resolution, the semantic objects in question would be lists as well, as opposed to multisets, and as such, could be implemented in much the same way as they are in the phenogrammar.

5.2.1 Tectogrammar

In chapter 4 we added the additive disjunction type constructor $\oplus$. It will become useful to make the following notational addition of the $n$-ary coproduct type constructor $\bigoplus$, which maps $\oplus$ over a collection of types in the obvious manner:

$$\bigoplus A_1, \ldots, A_n = \text{def} A_1 \oplus \ldots \oplus A_n$$

\textsuperscript{2}Systems mixing unordered linear connectives with ordered linear connectives have been explored, e.g. the Partially Commutative Linear Logic of [dG96] and the Intuitionistic Noncommutative Linear Logic of [PP99].

\textsuperscript{3}We have in mind here something slightly different from our conception of strings as the quasi-base type $\text{St}$, which we will clarify shortly.
Likewise if we have a countable indexing set $J$, we write

$$\bigoplus_{i \in J} A_i$$

When $J$ is obvious from context it may be omitted. While it would be possible to implement this directly in the object language by adding dependent sum types ($\Sigma$-types), this would require (minimally) us to define the type of natural numbers, which is a complication outside the scope of the current inquiry. For the time being, we use $n$ to indicate a metavariable over natural numbers.

In the tectogrammar, we will want a type analogous to a multiset: a collection of expressions for which order is insignificant, but repeated elements are significant. We will define the multiset monad type constructor, which is analogous to the better-known Kleene star monad, but here, since $\otimes$ is commutative, these types denote multisets rather than strings. We will write the multiset monad type constructor $A^*$ for $A$ a metavariable over types, by analogy to the Kleene star. Intuitively, injection into a singleton multiset corresponds to the $\eta$ natural transformation of the monad, with multiset union corresponding to $\mu$.

We write $\otimes^n A$ for the $n$-fold tensor power of $A$, that is, $A \otimes A \otimes A \ldots$, $n$ times. That is, $\otimes^0 A = 1$, $\otimes^1 A = A$, $\otimes^2 A = A \otimes A$, etc.

Given this piece of convenient shorthand we can first define the nonempty multiset monad in the tectogrammar using the countable coproduct type constructor (where $n$ is a natural number):

$$A^+ = \text{def} \bigoplus_{n>0} \otimes^n A$$

that is,

$$A^+ = A \oplus (A \otimes A) \oplus (A \otimes A \otimes A) \ldots$$

We can then define the multiset analogue of Kleene star, which we write $A^*$ to be simply:

$$A^* = \text{def} 1 \oplus A^+$$
that is,

\[ A^* = \bigoplus_n^n A \]

or spelled out,

\[ A^* = 1 \oplus A \oplus (A \otimes A) \oplus (A \otimes A \otimes A) \ldots \]

which is easily shown to be correct by the fact that \( \otimes \) distributes over \( \oplus \):

\[
\begin{array}{c}
A \otimes (B \oplus C) - A \otimes (B \oplus C) \quad B \oplus C - B \oplus C \\
A = A \quad B = B \quad \oplus I \\
A, B = (A \otimes B) \oplus (A \otimes C) \quad \oplus I_1 \\
A, B \otimes C = (A \otimes B) \otimes (A \otimes C) \quad \oplus E \\
\end{array}
\]

As an immediate corollary of the definition of \( A^* \), it is obvious that \( A^* \rightarrow A^* \).

Ultimately, we will provide the following tectotype for \( \text{and} \): \( \circ A \rightarrow (\circ A)^+ \rightarrow \circ A \). This reports that \( \text{and} \) takes as its first argument a coordinate structure of \( A \)'s, followed by a multiset of coordinate structures of \( A \)'s, and yields a coordinate structure of \( A \)'s. The reader may reasonably ask two questions. First, why \( A^+ \) instead of \( A^* \)? While empty multisets may be convenient tools for building structure (in fact, we will make use of something similar in the phenogrammar), we certainly do not want to build coordinate structures out of empty lists. Secondly, why do the two monads in question (the coordination monad and the multiset monad) appear in the order in which they appear; that is, why \( (\circ A)^+ \) and not \( \circ (A^*) \)? Intuitively, this is because we want to allow the construction of lists of coordinate structures that themselves occur within coordinate structures. Consider for example the following, the iterated conjunct part of which can be given the tectotype \( (\circ \text{NP})^+ \), thus making the entire conjunction simply \( \circ \text{NP} \), as is desired:

\[
(68) \quad \text{a. Which couples are married?}
\]

Robert and Cersei, Joffrey and Margaery, and Tyrion and Sansa.

This stands in contradistinction to the type of coordinate structures made of lists, which would license constructions like \( \text{Robert, Joffrey, Tyrion, and Cersei, Margaery, Sansa} \) which we do not believe to be well-motivated for English.
5.2.2 Phenogrammar

There is a certain apparent redundancy to defining list types in a grammar formalism which already makes reference to strings. That is, with reference to our treatment of strings as a basic type, one might reasonably ask: strings of what?\(^4\) Up till now, we have taken strings to be (near)-primitives\(^5\) of the phenogrammar. Rather than redefine and extend this notion, we will continue to treat strings as purely phenogrammatical objects, and by contrast, we will treat lists as being objects definable in multiple logics, whose behavior is controlled by specific grammar rules (cf. the lack of grammar rules for strings). In summary, we distinguish between strings as the essential building blocks of phenogrammatical expressions, and lists, which are more complex. It would of course be possible to give a uniform treatment to both, by stipulating that what we have called “strings” up till now are, say, lists of phonological words, which we then provide a type for, etc. Nevertheless, it will in no way be misleading to think of St as a base type, and lists as constructed.

We use brackets \([\ ]\) ambiguously between type constructors and term constructors. For a given type \(A\), we add the type of lists of \(A\)s, written \([A]\). The empty list whose underlying type is \(A\) is also written \([A]_A\), and we will define some syntactic sugar making use of brackets momentarily. First, we give the (single-component) logical rules governing lists:

\[
\vdash [A] : [A] \quad [\ ]
\]

This rule simply asserts that the empty list \([A]_A\) is a list of \(A\)s.

\[
\vdash x : A \quad \vdash l : [A] \\
\vdash (\text{cons}_A x \ l) : [A]_A \quad [I]
\]

The function \(\text{cons}_A\) allows us to construct non-empty lists of \(A\)s. When it is clear from context what \(A\) is, as it typically will be, we may omit the subscript on \(\text{cons}\). In general,

\(^4\)Although it is a little odd to ask such questions about base types, which are simply stipulated.

\(^5\)Strings are not primitives as such, since they are defined in terms of the “monoid type” \(m\).
the first step to constructing lists will be to \texttt{cons} a particular item to the empty list [], thus providing a natural base for the recursive construction and destruction of lists.

As mentioned previously, we will overload the [] constructor for the sake of legibility, using it as both a term constructor and a type constructor. To that end, we make the following abbreviation (with $\overline{x} = \text{def } x_0, x_1, \ldots, x_n$):

$$[a] = \text{def } (\text{cons } a [])$$
$$[a, \overline{x}] = \text{def } (\text{cons } a [\overline{x}])$$

While it is possible to give an elimination rule for []-types, it requires the definition of several additional functions in pheno\textsuperscript{6}, and we do not currently see any particular linguistic motivation for doing so.

\section*{Revising \texttt{say}}

Since we have added new types to the phenogrammar, and we intend to use them with coordinate structures, we will need to augment the definition of the \texttt{say} function with a couple of additional clauses. Remembering that the entire purpose of \texttt{say} is to obtain the string support from a particular phenoterm, these are not terribly complicated. If the list in question is the empty list, we simply return the empty string $\epsilon$. If the list is non-empty, we apply \texttt{say} to the first element of the list, and then concatenate that element with whatever the result of applying \texttt{say} to the rest of the list is. We add the following tail-recursive clauses to the existing definition of \texttt{say}:

$$\texttt{say}_{[A]} []_A = \text{def } \epsilon$$
$$\texttt{say}_{[A]} (\text{cons } a \ell) = \text{def } (\texttt{say}_{[A]} a) \cdot (\texttt{say}_{[A]} \ell)$$

$^6$e.g. head, tail, map, concat, append, foldr, etc.
In adding to the definition of $\text{say}$, we choose here to define this function by cases, rather than resorting to an explicit $\text{case...in...}$ term constructor. Such a definition remains unproblematic, since this style is equivalent to using a coproduct type together with injections. For the time being we have chosen not to make this connection explicit, for the sake of expository clarity.

Continuing, by way of example, we evaluate the application of $\text{say}$ to the list corresponding to the pheno of the expression $\text{choked, died}$, which we represent in pheno as $
 
 = [([\downarrow, (i \text{ CHOKED}))], ([\downarrow, (i \text{ DIED}))] : [(\text{St}_{1})]] \text{ as follows, remembering that } i \text{ is short for the phénominateur } \lambda s.s \cdot v, \text{ which appears already applied and reduced in the calculation below:}$

\[
\begin{align*}
\text{say}_{[(\text{St}_{1})]} & (\text{cons} (\downarrow, \lambda s.s \cdot \text{CHOKED}) \cdot (\text{cons} (\downarrow, \lambda t. t \cdot \text{DIED}) \cdot \text{[]}_{(\text{St}_{1})})) \\
& = (\text{say}_{(\text{St}_{1})} (\downarrow, \lambda s.s \cdot \text{CHOKED})) \cdot (\text{say}_{[(\text{St}_{1})]} (\text{cons} (\downarrow, \lambda t. t \cdot \text{DIED}) \cdot \text{[]}_{(\text{St}_{1})})) \\
& = (\text{say}_{(\text{St}_{1})} (\downarrow, \lambda s.s \cdot \text{CHOKED})) \cdot (\text{say}_{(\text{St}_{1})} (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = (\text{say}_{\text{St}_{1}} (\downarrow, (\lambda s.s \cdot \text{CHOKED})) \cdot (\text{say}_{(\text{St}_{1})} (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = (\text{say}_{\text{St}_{1}} \lambda s.s \cdot \text{CHOKED}) \cdot (\text{say}_{(\text{St}_{1})} (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = (\text{say}_{\text{St}_{1}} \lambda s.s \cdot \text{CHOKED}) \cdot (\text{say}_{(\text{St}_{1})} \cdot (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = (\text{say}_{\text{St}_{1}} \epsilon \cdot \text{CHOKED}) \cdot (\text{say}_{(\text{St}_{1})} \cdot (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = (\text{say}_{\text{St}_{1}} \text{CHOKED}) \cdot (\text{say}_{(\text{St}_{1})} \cdot (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = (\lambda s.s \text{ CHOKED}) \cdot (\text{say}_{(\text{St}_{1})} \cdot (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = \text{CHOKED} \cdot (\text{say}_{(\text{St}_{1})} \cdot (\downarrow, \lambda t. t \cdot \text{DIED})) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = \text{CHOKED} \cdot (\text{say}_{\text{St}_{1}} \cdot (\downarrow, (\downarrow, \lambda t. t \cdot \text{DIED}))) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})}) \\
& = \text{CHOKED} \cdot (\text{say}_{\text{St}_{1}} \lambda t. t \cdot \text{DIED}) \cdot (\text{say}_{[(\text{St}_{1})]} \cdot \text{[]}_{(\text{St}_{1})})
\end{align*}
\]
\[ = \text{CHOKED} \cdot (\text{say}_{\text{St}} \lambda t. \text{T} \cdot \text{DIED} \text{vac}_{\text{St}}) \cdot (\text{say}_{\{\text{sT}_{1}\}} \bigcup_{\{\text{sT}_{1}\}}) \]
\[ = \text{CHOKED} \cdot (\text{say}_{\text{St}} \lambda t. \text{T} \cdot \text{DIED} \epsilon) \cdot (\text{say}_{\{\text{sT}_{1}\}} \bigcup_{\{\text{sT}_{1}\}}) \]
\[ = \text{CHOKED} \cdot (\text{say}_{\text{St}} \epsilon \cdot \text{DIED}) \cdot (\text{say}_{\{\text{sT}_{1}\}} \bigcup_{\{\text{sT}_{1}\}}) \]
\[ = \text{CHOKED} \cdot (\lambda s. \text{s DIED}) \cdot (\text{say}_{\{\text{sT}_{1}\}} \bigcup_{\{\text{sT}_{1}\}}) \]
\[ = \text{CHOKED} \cdot \text{DIED} \cdot (\text{say}_{\{\text{sT}_{1}\}} \bigcup_{\{\text{sT}_{1}\}}) \]
\[ = \text{CHOKED} \cdot \text{DIED} \cdot \epsilon \]
\[ = \text{CHOKED} \cdot \text{DIED} \]

5.2.3 Grammar rules

Since this is part of the grammar writ large, we need to specify grammar rules tying together the disparate components. First, we have the rule we call \textbf{Cons} (for construct), not to be confused with the phenogrammatical function \textit{cons}, which essentially adds pheno-expressions to a list and tectotypes to the multisets.

\[
\Gamma \vdash s : S \quad \Delta \vdash l : [S] : A^* \quad \rightarrow \quad \Gamma, \Delta \vdash (\text{cons} \ s \ l) : [S] : A^+ \quad \text{Cons}
\]

This rule says that if you have a sign whose pheno is \(s\), of phenotype \(S\), and you have a phenogrammatical list \([S]\) called \(l\), then you can \textit{cons} an \(s\) onto \(l\), and the result will also be the (nonempty) list \((\text{cons} \ s \ l)\) of type \([S]\). In the tectogrammar, this corresponds to using multiplicative conjunction \(\otimes\) between a multiset type \(A^*\) and its corresponding underlying type \(A\), which results in a nonempty multiset type \(A^+\). Since \(\vdash A^+ \rightarrow A^*\), we will generally make use of a slightly derived version of this rule whose output tectotype is \(A^*\) rather than \(A^+\).

\[
\vdash [[] : [S] : 1]
\]
This rule states that in pheno, there is an empty list for each type $S$, obviously of type $[S]$. In the tectogrammar, this is represented as the multiplicative (tensor) identity $1$. As an immediate consequence of the rules given in chapter 4 for additive disjunction in the tectogrammar, taken together with the definition of $A^*$ (as $1 \oplus A^*$), we have the following derived rule, which states that the phenogrammatical empty list is itself a multiset (although the empty multiset) in the tectogrammar:

$$\Gamma \vdash [\ ] : [S] ; A^*$$

Finally, in derivations, we customarily omit instances of this rule as it is easily recognizable. Instead we will write the following (derived and abbreviated) rule for cases of the Cons-rule involving the empty list $[\ ]$:

$$\Gamma \vdash s : S ; A \quad \Gamma \vdash [s] : [S] ; A^*$$

Cons

Even when we do make this rule explicit, that is, in cases where the tail of the list to be constructed is non-empty, we will still customarily omit the labeling of the inference rule, as it is easily recognizable from the phenoterm constructors. The categorically inclined reader will note that this corresponds to the $\eta$ natural transformation of a monad: the list monad in pheno, and the multiset monad in tecto. We note in passing that the presence of these rules implies that non-iterated coordination can be analyzed as a special case of iterated coordination, where the initial conjunct is simply a unary iteration.

5.3 Analysis

5.3.1 Overall strategy

Now that we have the ability to construct iterated expressions of like type, most of the work is complete. Having added the necessary clauses to the definition of say, we are now in a position to revise the lexical entry schema for and. We write this to take the non-iterated
conjunct first, and the iterated (left) conjunct second, but we believe this choice to be without theoretical import.

\[ \vdash \lambda c_2 : A_\phi . \lambda c_1 : [A_\phi] . \downarrow_\phi \left( (\text{say}_{A_\phi} c_1) \cdot \text{AND} \cdot (\text{say}_{A_\phi} c_2) \right) \]

: \quad A_\phi \rightarrow [A_\phi] \rightarrow A_\phi

; \quad \triangledown B \rightarrow (\triangledown B)^+ \rightarrow \triangledown B

The only difference between this version and the prior version is that the second conjunct is typed as a list in the phenogrammar, and a multiset in the tectogrammar, and correspondingly, the version of \textit{say} we apply to the phenogrammatical argument in question has a different type. Other than that, the strategy is essentially the same: take one list conjunct whose elements are under the image of a particular phenominator, take another conjunct which is under the image of the same phenominator, \textit{say} both of them, put the string \textit{AND} in between them, apply the phenominator to the result, and downcast that back into the relevant subtype.

5.3.2 Derivation of selected examples

Iterated string coordination

We first consider the simplest case, given in example (11), where the conjuncts are strings in the phenogrammar, and noun phrases in the tectogrammar.

Observe that the tectotype $\triangledown \text{NP} \rightarrow \text{NP} \rightarrow \triangledown \text{S}$ is obtainable from the tectotype of transitive verbs $\text{NP} \rightarrow \text{NP} \rightarrow \text{S}$ by the following proof (tecto only, for illustrative purposes, and with permutation implicit):

1. 

\[
\begin{array}{ll}
\text{NP} \rightarrow \text{NP} \rightarrow \text{S} \rightarrow \text{NP} \rightarrow \text{S} \rightarrow \text{NP} \rightarrow \text{NP} & \rightarrow E \\
\text{NP} \rightarrow \text{NP} \rightarrow \text{S}, \text{NP} \rightarrow \text{NP} \rightarrow \text{S} & \rightarrow I \\
\text{NP} \rightarrow \text{NP} \rightarrow \text{S}, \text{NP} \rightarrow \triangledown (\text{NP} \rightarrow \text{S}) & \rightarrow \triangledown I \\
\text{NP} \rightarrow \text{NP} \rightarrow \text{S}, \text{NP}, \text{NP} \rightarrow \triangledown S & \rightarrow \triangledown \triangledown I \\
\triangledown \triangledown \text{NP}, \text{NP} \rightarrow \text{NP} \rightarrow \text{S}, \text{NP}, \text{NP} \rightarrow \triangledown \triangledown S & \rightarrow \triangledown \triangledown \triangledown I \\
\text{NP} \rightarrow \text{NP} \rightarrow \text{S}, \text{NP} \rightarrow \triangledown \text{NP} \rightarrow \text{NP} \rightarrow \triangledown S & \rightarrow \triangledown \triangledown \triangledown \triangledown I \\
\text{NP} \rightarrow \text{NP} \rightarrow \text{S}, \text{NP} \rightarrow \triangledown \text{NP} \rightarrow \text{NP} \rightarrow \triangledown \triangledown S & \rightarrow \triangledown \triangledown \triangledown \triangledown \triangledown I \\
\end{array}
\]
So we can justifiably assign that tectotype to the verb *hated*, which is indicated in the full derivation below by \(\vdash\), vertical ellipsis. Now we can continue with the derivation of the example in question. First, we derive the first conjunct:

2.

\[
\frac{\vdash \text{Cersei} : \text{St}; \text{NP}}{\vdash \text{Cersei} : \text{St}; \circ \text{NP}} \quad \text{Ret} \quad \vdash \downarrow_n \text{Cersei} : \text{St}_n; \circ \text{NP}
\]

First, we inject the conjunct into the coordination monad type over noun phrases \(\circ \text{NP}\).

Now, since the string *Cersei* is trivially under the \(n\) phenominator \(\lambda s.s\), \(P_n\) holds, allowing us to downcast into the relevant subtype. Next, we construct the second conjunct:

3.

\[
\frac{\vdash \text{Joffrey} : \text{St}; \text{NP}}{\vdash \text{Joffrey} : \text{St}; \circ \text{NP}} \quad \text{Ret} \quad P_n \quad \vdash \downarrow_n \text{Joffrey} : \text{St}_n; \circ \text{NP}
\]

\[
\frac{\vdash \text{Tywin} : \text{St}; \text{NP}}{\vdash \text{Tywin} : \text{St}; \circ \text{NP}} \quad \text{Ret} \quad P_n \quad \vdash \downarrow_n \text{Tywin} : \text{St}_n; \circ \text{NP}
\]

\[
\vdash [\downarrow_n \text{Joffrey}, \downarrow_n \text{Tywin}] : [\text{St}_n]; (\circ \text{NP})^+ \quad \text{Cons}
\]

For both *Joffrey* and *Tywin*, the strategy is at first the same as it was for *Cersei*. We inject each conjunct into the coordination monad, then downcast into the subtype. At this point, things become slightly more complicated, as we inject the downcast version of *tywin* using the modified Cons rule described earlier.\(^7\) Then, the two names are combined to form the iterated conjunct, which is represented in the phenogrammar by the list type \([\text{St}_n]\), and in the tectogrammar by the multiset type \((\circ \text{NP})^+\). We instantiate the lexical entry schema for *and* with \(A = \text{St}, \varphi = n,\) and \(B = \text{NP}\), resulting in the following:

\[
\vdash \lambda c_2 : \text{St}_n, \lambda c_1 : [\text{St}_n]. \quad \downarrow_n (n ((\text{say}_{[\text{St}_n]} c_1) \cdot \text{AND} \cdot (\text{say}_{\text{St}_n} c_2)))
\]

\[
: \quad \text{St}_n \rightarrow [\text{St}_n] \rightarrow \text{St}_n
\]

\[
: \quad \circ \text{NP} \rightarrow (\circ \text{NP})^+ \rightarrow \circ \text{NP}
\]

The full derivation proceeds as follows:

\(^7\)I.e. to be entirely precise, there should be a step showing the cons-ing of the conjunct to the empty list, but we omit this for clarity and space.
First we combine *and* with its two conjuncts to produce the string *Joffrey* · *Tywin* · *and* · *Cersei*, which is a coordinated noun phrase in the tectogrammar. We upcast this into the supertype St. Due to the equivalence stated earlier, *hated* is suitable to combine with the coordinated NP object, and then its subject *Tyrion*, resulting in the desired expression, which is a string in the phenogrammar and a sentence exhibiting coordination in the tectogrammar.

**Iterated functor coordination**

Functor coordination as exemplified by (12) presents little problem, as the definition for *say* has been extended in such a manner as to handle lists of functors in essentially the same way as lists of strings.

With the cotensor strength lemma (CTS), it follows trivially and directly that

![Inference](image)

We use this equivalence in an abbreviated form in the proof below, where we label it CTS2, intended to be mnemonic for “cotensor strength, a couple of times”. We turn now to the derivation of the first conjunct:

1. ![Inference](image)

As before, we inject the conjunct into the coordination monad, and then downcast into the functional subtype (St₂), since transitive verbs are provably under the phenominator t. Now we continue with the derivation of the second conjunct:
together to form a list / multiset. We instantiate the lexical entry schema for \( \text{and} \) and \( \text{cons} \) with a coordination example. We simply inject and downcast the relevant conjuncts, and then \( \text{cons} \) them together to form a list / multiset. We instantiate the lexical entry schema for \( \text{and} \) with \( A = \text{St}_2 \), \( \varphi = t \), and \( B = \text{TV} \), yielding the following lexical entry:

\[
\vdash \lambda \text{c}_2 : (\text{St}_2)_t, \lambda \text{c}_1 : [(\text{St}_2)_t]. \downarrow (t ((\text{say}_{(\text{St}_2)_t} c_1) \cdot \text{AND} \cdot (\text{say}_{(\text{St}_2)_t} c_2))) \\
: (\text{St}_2)_t \rightarrow [(\text{St}_2)_t] 
\]

Finally, we can derive the entire sentence as follows:

\[
\vdash \lambda \text{c}_2 : (\text{St}_2)_t, \lambda \text{c}_1 : [(\text{St}_2)_t]. \downarrow (t ((\text{say}_{(\text{St}_2)_t} c_1) \cdot \text{AND} \cdot (\text{say}_{(\text{St}_2)_t} c_2))) \\
: (\text{St}_2)_t \rightarrow [(\text{St}_2)_t] 
\]

The strategy here is the same in essence as above, except for the fact that in this case the primary verb phrase functor is itself a coordinate structure. First we combine \( \text{and} \) with its conjuncts, then upcast the result to restore its ability to combine with its arguments in the phenogrammar. Next, we make use of cotensor strength to restore its ability to combine with its arguments in the tectogrammar. From this point things proceed straightforwardly, resulting in the pheno string \( \text{OLENNA} \cdot \text{HATED} \cdot \text{POISONED} \cdot \text{AND} \cdot \text{KILLED} \cdot \text{JOFFREY} \), which is asserted to be a sentence exhibiting coordination in the tectogrammar.
Iterated coordination with unlike category coordination

Of course it is possible to combine unlike category coordination with iterated coordination, as indicated in example (13).

We make the assumption that deceased is a predicative adjective (tectotype PrA), and a string phenogrammatically, which we take to be reasonably uncontroversial. We remind the reader of the lexical entries for the expressions from chapter 4, and add one for deceased:

\[
\begin{align*}
\vdash \text{evil} &: \text{St}; \text{PrA} \\
\vdash \text{deceased} &: \text{St}; \text{PrA} \\
\vdash \text{a \cdot tyrant} &: \text{St}; \text{PrN} \\
\vdash \text{became} &: \text{St}_2; (\text{PrA } \oplus \text{PrN}) \rightarrow \text{VP}
\end{align*}
\]

We can instantiate the lexical entry schema for and with \( A = \text{St}, \varphi = n, \) and \( B = (\text{PrA } \oplus \text{PrN}) \) to yield the following:

\[
\begin{align*}
\vdash \lambda c_2 : \text{St}_n. \lambda c_1 : [\text{St}_n]. \downarrow_n (((\text{say}_{\text{St}_n} c_1) \cdot \text{AND} \cdot (\text{say}_{\text{St}_n} c_2))) \\
: \quad \text{St}_n \rightarrow [\text{St}_n] \rightarrow \text{St}_n \\
; \quad \circ (\text{PrA } \oplus \text{PrN}) \rightarrow (\circ (\text{PrA } \oplus \text{PrN}))^+ \rightarrow \circ (\text{PrA } \oplus \text{PrN})
\end{align*}
\]

The full derivation appears on the following page. We abbreviate the entire lexical entry instantiation for and with just its string support \text{AND}, owing to its unwieldy size and presence above. The reader should also note in passing that type annotations in lambda terms have been suppressed where they are clear from context, and the labels for inference rules appear atypically on the left-hand side, in order to fit within the bounds of the page.

This derivation simply combines strategies from chapter 4 with the current chapter. Most of the proof is taken up by injection bookkeeping, ensuring that each of the conjuncts is first underneath the appropriate disjunction type, and subsequently underneath the coordination monad. The functoriality lemma allows the verb become to combine with the
Figure 5.1: Derivation for *Joffrey became evil, a tyrant, and deceased*
coordinated predicative, and cotensor strength allows the result to combine with a simple NP subject Joffrey. The result is, as ever, a string in the phenogrammar, and a sentence which possibly contains a coordinate structure in the tectogrammar.

5.4 Discussion

We have shown empirical motivation for the addition of iterated types to grammar formalisms in general. We have discussed several proposals for doing so, and shown that they can be brought to bear on developing one in LCG_ϕ. We show how to add list types to the phenogrammatical component and multiset types to the tectogrammatical component. This requires expanding some of the functions defined along the way in order to analyze coordination. We provide inference rules for each component, and grammar rules linking the components. We have revised our lexical entry for the coordinating conjunction and and shown how these changes taken together can be used to analyze cases of iterated string coordination, iterated functor coordination, and finally iterated unlike category coordination. We turn now to a discussion of some of the outstanding issues and interesting avenues for future research.
Chapter 6

Discussion and Future Work

6.1 Future work

6.1.1 Linear Categorial Grammar on the whole

Linear Categorial Grammar has been able to successfully analyze a broad range of English phenomena. It has had additional success with Serbo-Croatian in [Mih12], and has been more modestly extended to cover data from Chinese [Dua15], with work on K’iche’ ongoing. It would be wonderful to extend LCG to a much broader range of data from other languages. LCG’s curryesque architecture is a perfect sandbox in which to explore languages with more complex morphology than English. In particular, the ability to construct lists polymorphically in theory allows for any number of nested levels of concatenation, suggesting ways to come to grips with cliticization, affixation, reduplication, and the like.

Furthermore, the fact that LCG has a functional phenogrammar articulated in higher order logic together with the fact that the interface between pheno and tecto is relational rather than functional allows for the potential exploration of prosody as a first-class citizen in its own right, from the perspective of a categorial grammarian. It is possible to write lexical entries involving pitch accents, boundary tones, downstep, and the like, meaning that it is possible to associate both combinatorics and meaning with tunes and other suprasegmental representations. This opens up a huge array of phenomena to analysis: from alternative questions to focus to disambiguation, LCG in theory lets us begin to come to grips with the ways in which these interact with segmental representations in a formal setting.
6.1.2 Coordination in LCG

Empirically speaking, there are a number of coordination phenomena for which we have not provided analyses. In some case, this is owing to the fact that the conjuncts are not under the image of a phenominator. These include but are not limited to examples such as the following:

(69) Hodor guarded Bran and Brienne, Jaime. (Gapping)
(70) This is the prince who Cersei loves but Tyrion thinks is sadistic. (non-parallel / medial ATB extraction)

We are convinced that gapping is restricted enough a phenomenon that the current coordination lexical entry need not be revised in order to account for it. Instead, we have in mind an analysis similar to that found in [KL12] and [KL15c], albeit recoded in LCG_ϕ. In parallel cases, LCG_ϕ is theoretically capable of handling “across the board” (ATB) extraction, since each conjunct will be under the image of the same phenominator. The difficulty of accounting for the non-parallel cases is compounded by the fact that other constructions (e.g. parasitic gaps) seem to share a similar character, and so we will need a more general method of addressing this issue.

Encoding other grammars

We would like to be able to show that the Lambek Calculus can be encoded in LCG, and we have developed an experimental algorithm which deduces phenominators from directional types (in the Lambek sense) that in theory enables this translation, but the details are not sufficiently worked out to be able to include it here. Similarly, we believe it is possible to encode Hybrid Type Logical Categorial Grammar (HTLG) in LCG as well, once the Lambek translation has been achieved. Furthermore, it has been shown in [Moo15] that (at least) both HTLG and the Discontinuous Lambek Calculus can be embedded into first
order multiplicative linear logic (MILL1). The relationship between LCG and MILL1 is currently unclear.

**The coordinate structure constraint**

As currently stated, our system overgenerates in that it runs afoul of the so-called “coordinate structure constraint” of [Ros67], which holds roughly that if an extraction takes place from a coordinate structure, every conjunct must contain a gap corresponding to the type of the extracted element. We have no way of enforcing this condition on extraction. As long as the phenotypes of the conjuncts are under the image of the same phenominator, they are conjoinable in LCG_\(\varphi\). So structures like the following, consisting of the conjunction of a transitive verb with an object gap and an intransitive verb in principle become licenseable:

(71) This is the wine that Tyrion drank and fell asleep.

We do not presently feel that there is substantial evidence supporting the CSC as a syntactic constraint. Instead we follow [Keh96] and [KL15a] in assuming that the patterns apparently exhibited with respect to the CSC follow more generally from pragmatic principles.

**On metalogical encoding and decidability**

It is in theory possible to encode the structure of the framework in the object language of a more powerful type-theoretic formalism of sufficient expressivity. That is, in order to encode the axioms and rules of LCG and LCG_\(\varphi\), we need the ability to assert judgments about when terms and types are well-kinded, and the ability to write expressions representing LCG’s grammar rules and the reductions and conversions associated with making inferences in each component.

Encoding phenominators is tricky. However, since it is possible to encode higher order logic in the Edinburgh Logical Framework (LF) [HHP93], it ought to be possible to encode
Lambek and Scott-style subtypes in LF as well. Since the predicates we are after are ultimately expressed in terms of phenominators, and in the general case, phenominators are encoded as linear combinators (with a normal form), it will be necessary to encode linear logic as well. Fortunately, this is also possible, as indicated by [Cra10]. The fact that phenominators can themselves contain variables of fine-grained types adds recursion to the system, however, and this provides a significant complication.

There are certain aspects of the formal foundations of $\text{LCG}_\phi$ which deserve closer study. We are particularly concerned about decidability in two forms. Generally speaking, it is well-known that the addition of Lambek and Scott-style subtyping makes proof search undecidable, since it essentially hinges on deciding whether two functions are equal. It is our hope that this can be avoided due to the fact that we do not make global use of subtyping, but only with respect to the very small class of phenominators we need to use in order to analyze coordination. Second, the related smaller problem of deciding whether a given term is under the image of a particular phenominator. Due to the encoding of strings provided, if we exclude the term constructor $\uparrow$ from the definition of phenominators, then phenominators become linear combinators in the sense of having no constants whatsoever. We believe that given this restriction, the phenominator decision problem is decidable.

However, the question of decidability when $\uparrow$ is added back into the system, as it seems it must be to analyze ACC, is one which is still open. It is our suspicion that since phenotypes ultimately bottom out in string types (always under the phenominator $n$), the procedure for determining whether a given expression is under the image of a phenominator is decidable, though such a proof currently remains regrettably out of reach. While encoding the metatheory of LCG is complicated, type theories invoking dependent types are a crucial umbrella under which we may make an attempt. Furthermore, the encoding of the lax modal $\bigcirc$, which we use for our analysis of unlike category coordination, has also been explored in this context in [PD01] and [Sim14]. Encoding LCG in one of the tools designed
for this purpose\textsuperscript{1} is a potential area for future research.

6.1.3 Unlike category coordination in LCG

The first and most notable lacuna is the absence of a genuine semantics for the analysis given here. Fortunately, it is not difficult to do this, and it can be accomplished by the addition of meaning postulates defining in detail what the precise articulation of the constants we give is to be. For a proposal regarding how this may be accomplished, the reader is referred to [Mar13] and [MP14], which describe the framework of Dynamic Categorial Grammar (DyCG), which is in essence LCG with a dynamicized (and far more articulated) theory of semantics, with a discussion of static semantics along the way.

While unlike category coordination generally refers to the kinds of coordinate structures described above, there is another way in which ostensibly dissimilar categories can be coordinated, namely that of asymmetrical or non-parallel coordination of “gappy” expressions within a relative clause. The acceptability of such constructions has been debatable, historically, though to our ears, examples like the following sound impeccable:

(72) Joffrey is a king who Cersei loves and Tyrion thinks is a tyrant

The generalization here would seem to be that in fact, one can coordinate functors imposing asymmetrical phenogrammatical structure, as long as they will not receive their arguments. That is, phenogrammatically dissimilar functors may be coordinated just in case their “missing” arguments are inaudible, as is the situation with relative clauses. Since the lexical entry for and makes explicit reference to the fact that phenogrammatical conjuncts must be in the same phenominator subtype, such constructions are ruled out, and so the grammar undergenerates. Interestingly, many Curryesque grammars based on linear logic would have no trouble whatsoever generating this sentence, since no overt reference is made to functor “directionality”, but such grammars overgenerate wildly.

\textsuperscript{1}e.g. Twelf [Pfe91], Coq [CH84], or Agda [Nor07]
Since the only way two expressions may be coordinated is if they are under the image of the same denominator, and asymmetrical functors are **not** under the image of the same denominator, it would seem that they are uncoordinable. However, if both contain hypotheses sufficient to place the resulting expressions under the image of the same denominator, then they can be coordinated, pushing the problem into how to handle the fact that we have now made use of two hypotheses, and, and least with the case of relative clauses, we should in principle only have one. Linear logic again suggests a way out: the modal ! (pronounced “bang” or “of course”), which allows for the controlled reintroduction of the structural rules of weakening, and more importantly, contraction. But ! is too strong; we only need contraction, and reintroducing weakening to the system will allow for the proliferation of a host of spurious dependencies, and wanton overgeneration.

In [Mor94], Morrill suggests the addition of a connective ¡, much like the ! modality of linear logic. Whereas ! is used to reintroduce weakening and contraction, Morrill suggests that ¡ should allow only contraction, and then proceeds to sketch an analysis of parasitic gaps under relative clauses. A similar strategy could be pursued here, albeit for a situation where multiple gaps arise by virtue of coordination, rather than subordination. Since there are no actual constants of type ¡A (for any A), the reader should think of ¡ as enforcing the presence of a gap which is never intended to be filled. First, we assume that ¡ comes equipped with the following rules, extended in a manner after [BBdPH92]:

\[
\frac{\Gamma, \Delta \vdash s : S; ¡A; m : M}{\Gamma \vdash s : S; A; m : M} \quad \text{(D)ereliction}
\]

The **(D)ereliction** rule asserts that a resource of tectotype ¡A may be used as a resource of type A with no change to the phenogrammar or the semantics. It is worth noting that this corresponds to the counit natural transformation (sometimes written ϵ) of a comonad.

\[
\frac{\Gamma \vdash s : S; ¡A; m : M \quad \Delta, x : S; ¡A; y : M, x' : S; ¡A; y' : M \vdash t : T; B; n : N}{\Gamma, \Delta \vdash (\lambda x. \lambda x'. t) \ s \ s : T; B; (\lambda y. \lambda y'. n) \ m \ m : N} \quad \text{C}
\]

---

\[2\text{We provide only grammar rules here, and refer the reader to } [\text{Mor94}] \text{ for Gentzen sequent rules analogous to tectogrammatical inference rules for ¡. Furthermore, we omit the rule of Promotion, as it is cumbersome and does not appear to be relevant for the case under examination. See } [\text{BBdPH92}] \text{ for details.}\]
The rule of **(C)ontraction** states that if you can produce a resource of tectotype \( \text{i}A \), then you can use it within another resource of tectotype \( B \), relying on two resources of tectotype \( \text{i}A \), by copying instances of the relevant terms \( s \) and \( m \) for the variables used within the phenogrammar and semantics of the second premiss, respectively. To put it another way: since \( \text{i}A \) can be thought of as producing any number of resources \( A \), then one can reuse variables in the phenogrammar and semantics as many times as desired. The reader should note that this potentially destroys the linearity of the terms in question.

Furthermore, we can provide the following lexical entry for the relative pronoun *who* (ignoring the semantic component):

\[
(73) \quad \vdash (\text{rel who}) : \text{St}_1 \rightarrow \text{St}_1; (\text{iNP} \rightarrow \text{S}) \rightarrow \text{N} \rightarrow \text{N}
\]

Now, the central idea is that even though the functional conjuncts representing *Cersei loves ___* and *Tyrion thinks ___ is a tyrant* are not parallel with respect to their induced phenominator,\(^3\) and thus are incompatible for conjunction, there is a way we can coerce them to be compatible. If we hypothesize string variables in the relevant positions, then we can get each conjunct to be a string (containing a string hypothesis somewhere within it). Since the \( \text{C} \) rule will let us contract any number of expressions of tectotype \( \text{i}A \), then we stipulate that the tectotype of each of our hypothetical variables should be \( \text{iNP} \). Once we obtain the version of each conjunct whose phenogrammatical representation is a string, we can conjoin them under the \( n \) phenominator, leaving the two hypotheses unbound in the context. However, since they each have tectotype \( \text{iNP} \), we can contract on that variable. This leaves us with only one string variable in the context, bound twice within the body of the phenoterm. Withdrawing this variable leaves us with a term of phenotype \( \text{St}_1 \), and tectotype \( \text{iNP} \rightarrow \text{S} \), the right type to combine with the wh-relativizer *who*. We omit the entire proof for the sake of saving space, but it will ultimately be possible to derive the following sign:

---
\(^3\)In fact, the second conjunct is not under the image of a phenominator at all.
So the entire phrase is a phenogrammatical string, and a sentence potentially involving coordination in the tectogrammar, as desired. The same strategy could be employed to analyze parasitic gap constructions such as the following, though we do not wish to provide a complete account here:

(74) Which Stark daughter did rivals of ___ capture ___?

6.1.4 Iterated coordination in LCG

First and foremost, adding lists to the semantic component is a desideratum. These are somewhat trickier, since the semantics of coordination must be extended to cover list types, which means that the standard “generalized conjunction” of [PR83] will not work straight out of the box, a fact recognized in [KL15c], among others. Nevertheless, this does not seem to present a particularly enormous problem. The treatment of iterated coordination in [Sch05] makes a promising proposal, and we direct the interested reader to that work.

It is well-known that the naive introduction of recursion to the lambda calculus ultimately leads to inconsistency. Nevertheless, some notion of list-building seems inextricably tied to successfully analyzing coordination in English, so it must be our position that this at least appears to be a necessary evil. It is worth noting that [LS86] define their type theory with a natural number type, and basic arithmetic machinery. Given these, the Kleene star becomes internally definable, and this is worth exploring.

The issue of the coordinate structure constraint extends to iterated coordination as well. While it is generally taken to be the case that it is desirable to have a grammar such that one is forced in extraction contexts to ensure that every conjunct contains a gap, there is some evidence that this is not necessarily the case. Consider the following, after [Keh96]:

(75) This is the wine that Tyrion bought, came home, and drank in five minutes flat.
We find such examples to be marginally acceptable. The problem here is twofold: first, if such sentences are judged to be grammatical, then they violate the coordinate structure constraint. This is not a problem for LCG$_\varphi$ as it stands, since it has no way of enforcing the CSC to begin with. However, if we deem these to be ungrammatical, then we are similarly without a way to block their derivation.

6.2 Conclusions

In general, curryesque categorial grammars are desirable for the ease with which they handle phenomena which are nonlocal, or not explicitly directional, such as extraction, quantifier scope, and the like. However, coordination seems to be an example of a directionally-sensitive phenomenon without peer, and so curryesque grammars are ill-equipped to deal with it. Furthermore, for any grammar with linear logic\footnote{At least, without the exponentials ? and !.} as its centerpiece, coordination has an irretrievably nonlinear character, causing additional issues for such grammars. Linear Categorial Grammar has both of these properties, and suffers from both of these deficits.

However, we are able to extend LCG with several pieces of technology in order to restore what has been lost. First, we can add Lambek and Scott-style subtyping to the phenogrammar. We can articulate the notion of the phenominator using a typed lambda calculus, and we can use these two together to define subtypes sufficient to distinguish between expressions which are otherwise of like type. Once we can do this, writing a lexical entry to capture coordination with and is within our reach, though it necessitates the introduction of the function say (and vacuities along the way) as a left inverse for phenominators. With these technologies in place it is possible for LCG (extended to LCG$_\varphi$) to successfully analyze a wide spectrum of coordination cases, restoring what was lost in the move from Lambek grammars to curryesque grammars.
Furthermore, we can extend LCG with linear logic’s additive conjunction and additive disjunction connectives & and ⊕. This lets us build types expressing category neutrality of both the argument and functor varieties. By adding some nonlogical rules for handling predicatives, we show that functor neutrality can be given an analysis in LCG, and we provide proof-of-concept for analyzing argument neutrality in German. This places us on firm footing with respect to analyzing unlike category coordination.

We extend the grammar with the coordination monad type constructor ◯, providing a way to distinguish between ordinary categories and the slightly larger category of “categories potentially involving coordination”. Using the coordination monad together with additive disjunction allows for the analysis of UCC in LCG, and clarifies and remedies a number of outstanding issues with the preliminary version of such an analysis, given in [PH03].

It is subsequently possible to generalize additive disjunction, and to extend LCG with rules for linear logic’s multiplicative conjunction connective ⊗, which had been lurking implicitly in the background. We can use these two to define multiset types in the tectogrammar, and we can likewise add list types to the phenogrammar. We show how it is possible to extend say to cover expressions which are lists, and with all of these in place, it becomes possible to analyze iterated coordination in LCG as well. Of course, all of these various phenomena can interact, and we show that LCG handles such interactions in a natural way.

Natural language is rife with coordinate structures. Any grammar formalism worth its salt must be able to provide an analysis of coordination in its multivariate forms. Up till now there has been no obvious way for purely curryesque, linear logic-based grammars to do this. We show that by augmenting LCG with subtypes, monads, and a number of well-known connectives from linear logic, such an analysis can be given. Though much work remains to be done, it is exciting, and pleasing that LCG is shown to be a good candidate for articulating grammars, and the curryesque program remains viable.
References


169


