THE UNIVERSITY OF CHICAGO

LEARNING SYNTAX VIA DECOMPOSITION

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE HUMANITIES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF LINGUISTICS

BY

MARINA ERMOLAEVA

CHICAGO, ILLINOIS

JUNE 2021

Table of Contents

List of Figures

List of Tables

List of Definitions

List of Algorithms

Abbreviations

CFG context-free grammar

HoP hierarchy of projections

LI lexical item

MCFG multiple context-free grammar

MDL Minimum Description Length

MG minimalist grammar

SC small clause

SMC Shortest Move Constraint

Acknowledgments

First and foremost, I would like to thank the members of my dissertation committee. Without the mentorship of John Goldsmith and Greg Kobele, this dissertation would not have been possible; and their role in my development as a linguist cannot be overstated. I would also like to express my deep gratitude to Karlos Arregi, Jason Riggle, and Erik Zyman, for their insightful suggestions and the different perspectives they offered.

Outside UChicago, thanks to Khalil Iskarous, Roni Katzir, and Ezer Rasin for the interest they took in this project and for the helpful discussions we had. Looking back to earlier formative influences, I am greatly indebted to the professors at Moscow State University, who I worked with as an undergraduate student. Special thanks to Pavel Graschenkov, Mati Pentus, and Sergei Tatevosov, whose guidance was instrumental in setting me on this path.

On the personal front, I would like to thank Alëna Aksënova, my friend and co-author of many years. We've been through quite a few adventures together, beginning with our time as undergrads back in Moscow. Thanks as well to Daniel Edmiston, Jackie Lai, and Orest Xherija. I miss the all-linguist apartment we shared for two years and our debates about linguistics over beer and pizza. I also very much appreciate the rest of awesome people, UChicago affiliates and otherwise, who I was lucky enough to meet and spend time around. Thanks in particular to Arseniy Andreyev, Paolo Degiorgi, Liz Franson, and Sam Gray. Likewise, I am grateful to my social bubble of 2021, an incredible group of mathematicians and computer scientists, for helping me maintain work-life balance and for the healthy peer pressure to stay productive on the home stretch of this journey. A special shout-out goes

to Aygul Galimova and Connor Simpson for lending their support at the time when it was most needed.

Last but certainly not least, thanks to my parents. Part of the credit for this dissertation – and for anything I have ever accomplished – rightfully goes to them.

Abstract

Many patterns found in natural language syntax have multiple possible explanations or structural descriptions. Even within the currently dominant Minimalist theoretical framework (Chomsky 1995, 2000), it is not uncommon to encounter multiple analyses for the same phenomenon proposed in the literature. A natural question, then, is whether one could evaluate and compare syntactic proposals from a quantitative point of view. Taking this line of reasoning further, I aim to capture, formalize, and subsequently automate the intuition behind the process of developing a syntactic analysis.

The contributions of this dissertation are threefold. First, I show how an evaluation measure inspired by the Minimum Description Length principle (Rissanen 1978) can be used to compare accounts of syntactic phenomena implemented as minimalist grammars (Stabler 1997), and how arguments for and against a given analysis translate into quantitative differences. Next, I build upon Kobele's (2018, to appear) notion of lexical item decomposition to propose a principled way of making linguistic generalizations by detecting and eliminating syntactic and phonological redundancies in the data. Finally, I design and implement an optimization algorithm capable of transforming a naive minimalist grammar over unsegmented words into a grammar over morphemes. As proof of concept, I conduct a number of experiments on fragments of the English grammar, including the auxiliary system, passives, and raising verbs; l-selection of prepositional phrases; and allomorphy in verb stems. The experiments demonstrate how optimizing a quantitative measure can produce linguistically plausible analyses similar to those proposed in theoretical literature.

Chapter 1

Introduction

1.1 Discovering grammars

How do linguists construct analyses of syntactic phenomena? In more concrete terms, this question can be stated as the problem of producing a *grammar* – a finite description of a natural language's syntax – in a principled way, based on some set of sentences drawn from the language.

Different views can be found in the literature of what counts as an acceptable grammar and what is represented by the process of constructing one. Harris (1951) formalizes and catalogues linguistic research methods as explicit *procedures of analysis* in order to make precise the decisions that a linguist makes in the process of constructing a description of language data. These procedures draw generalizations from the *distribution* of linguistic elements, which is defined as the set of contexts in which they are found. Importantly, Harris' approach is not designed to produce a unique analysis of a given language phenomenon; two linguists using the procedures of analysis to work on the same data may arrive at different solutions. There does not have to be an optimal or best description, however that might be defined; although some may well be more convenient than others for specific purposes (Harris 1954, p. 148).

For Chomsky (1957), on the other hand, there exists a single correct grammar for every natural language, and the goal of a linguist is to find this unique grammar based on a given data set (corpus of utterances) drawn from the language in question. In this regard, the following three tasks are outlined, from the most to the least demanding:

- *discovery procedure*: a "practical and mechanical" way to obtain the correct grammar;
- *decision procedure*: a method of determining whether a given grammar is the best one;
- *evaluation procedure*: a method of determining which of two given grammars is better, given a data corpus.

While the discovery procedure is initially dismissed as unfeasible and impractical, a similar idea reappears, along with the evaluation procedure, in (Chomsky 1965). An *explanatory theory* of language is defined there as one capable of selecting a descriptively adequate grammar based on linguistic data. The components of such a theory mirror those of an *acquisition model*, or how a child learns a language, and are listed below:

i. a universal phonetic theory that defines the notion "possible sentence"

ii. a definition of "structural description"

iii. a definition of "generative grammar"

iv. a method for determining the structural description of a sentence, given a grammar

v. a way of evaluating alternative proposed grammars

(Chomsky 1965, p. 31)

The last requirement (v) is described as being twofold: it calls for a formal *evaluation measure*, some sort of quantitative indication of how good a grammar is, but also demands that the class of possible grammars be small enough so the evaluation measure can realistically select between them. In this framework, a precise and rich definition of "generative grammar"

serves to tighten the class of grammars. However, the theory still permits multiple grammars compatible with the same data set; the choice of grammar is under-determined by the language data alone. This is where the evaluation measure comes in: the correct grammar is the highest-valued one among those that describe the data correctly. Of course, exactly how to construct a reasonable evaluation measure is a major issue by itself. Chomsky and Halle (1968) make some concrete steps in this direction (for phonological rules), including a specific suggestion for the evaluation procedure based on rule length measured in symbols.

Chomsky's later work takes the idea of restricting what counts as a candidate grammar much further. By (Chomsky 1986), the description of a grammar has shifted away from rule systems and is split into two components: an innate universal system of principles and parameters and a language-specific lexicon of items defined by their phonological form and semantic properties, with the former getting most of the attention. Assuming a finite number of principles, parameters, and parameter values, the number of possible languages (apart from the lexicon) is also finite. This move sharply reduces the role of the evaluation measure or even dispenses with it altogether, as long as the universal grammar can be designed to permit only a single grammar compatible with the data.[1] The most recent and currently dominant iteration of generative grammar, the Minimalist Program (Chomsky 1995, 2000), continues this trend. Much of the system is assumed to be universal and innate, leaving no need or place for an evaluation measure; and language-specific properties that must be learned are largely shifted into the features of lexical items.

To summarize, the distinctions between approaches to grammar discovery mentioned so far boil down to the following:

- Harris (1951): the framework allows for multiple descriptions of a given language. Although some of them may be better for certain purposes, there is no explicit notion of the best grammar;

1. The *strong learning* approach of (Clark 2013, 2015) can be thought of as a formalization of this idea. For each set of strings, it requires the existence of a unique description called the *canonical grammar*. A strong learning algorithm is required to converge to this target grammar for each (formal) language.

- Chomsky (1965): the framework allows for multiple descriptions of a given language, one of which is the correct grammar, and these descriptions can be compared based on some quantitative measure;

- Chomsky (1986) and later work: the framework allows for a small number of descriptions of a given language, or even a single one; the correct grammar follows from the formal properties of the system and the language data. This stance can be considered a special case of the previous one, where the set of candidate grammars is sufficiently small to eliminate the need for an evaluation measure.

Approaches based on learning and those relying primarily on a rich, restrictive innate component are often thought of as mutually exclusive. However, as Katzir (2014) points out, the two can (and should) be reconciled. Different theoretical proposals often seem equally capable of capturing observed linguistic phenomena, necessitating an additional criterion to choose between them. At the same time, any theory of universal grammar already comes with a cognitively plausible evaluation measure, based on the principle of Minimum Description Length (Rissanen 1978) – as long as the grammars permitted by it are capable of *parsing*, or assigning structural descriptions to sentences as per (iv), and their description length can be compared.

In line with (Katzir 2014), this dissertation combines the learning focus of (Chomsky 1965) with the simplifying developments of the Minimalist approach, applying an evaluation measure to Minimalist lexical items. In what follows, I treat the definition in (i–v) as a rough roadmap. More specifically, it can be straightforwardly reworded as a *search problem*, defined by two primary components:

- a hypothesis space;

- a method of examining this space to find the best candidate.

In our case, (i–iv) define the hypothesis space of candidate grammars and (v) encompasses the task of navigating this space using the evaluation measure. The rest of this chapter

serves to clarify this approach: Section 1.2 narrows down the former, whereas Section 1.3 gives a high-level overview of the latter. Section 1.4 provides a detailed outline of the rest of the dissertation.

1.2 Encoding syntactic proposals

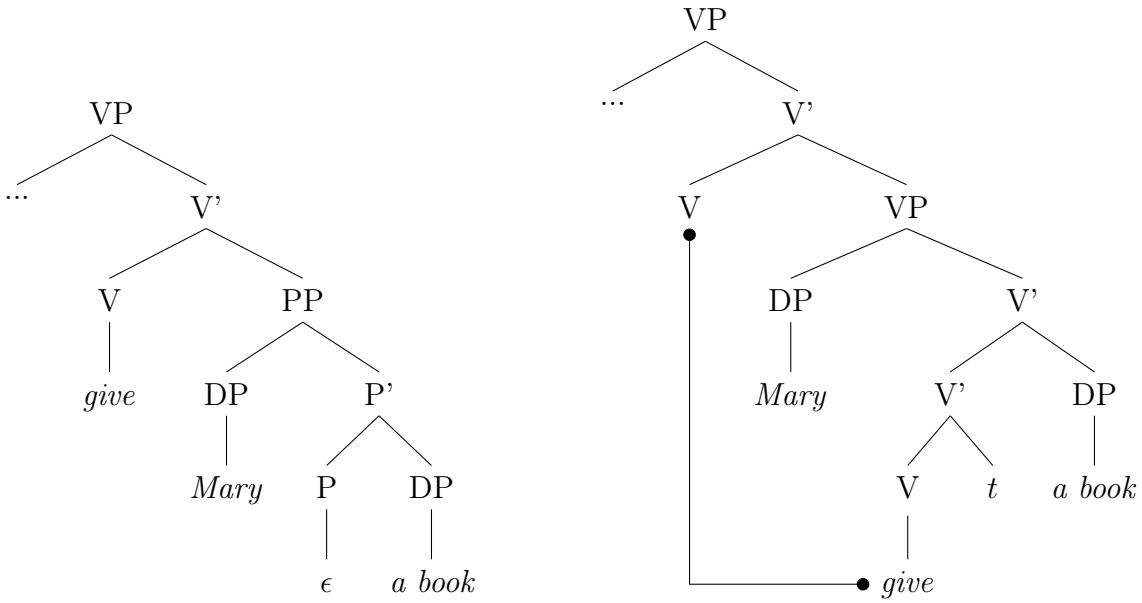
Marr's (1982) approach to understanding complex cognitive systems, including language, distinguishes between three levels of description:

- Computational: abstract specification of what the system computes;
- Algorithmic: structures representing the data and algorithms that manipulate them;
- Implementational: concrete realization of the algorithms in the hardware or wetware.

Johnson (2017) considers linguistic theories to be computational-level, while Peacocke (1986) places them at a "level 1.5", between the computational and algorithmic level. Syntactic literature in particular tends towards a big-picture outlook, abstracting away from algorithmic-level details such as full specifications of lexical items involved in derivations or syntactic features being checked by each application of a structure-building operation. At the same time, differences between competing analyses of the same phenomenon seem to fall closer to the algorithmic level.

For a concrete example, consider the double object construction (e.g. *John gave Mary a book*) in English (1.1). Any analysis of a syntactic phenomenon encodes two kinds of information: relatively theory-neutral, high-level facts that directly follow from the data, such as relations between words based on argument structure and linear order; and a proposed explanation of these facts – for instance, a specific configuration of lexical items constructed by structure-building operations. Descriptively, ditransitive verbs such as *give* appear in active sentences with three arguments: a subject, a direct object, and an indirect object. This

is (apparently) non-controversial. On the lower level,[2] disregarding the subject, one option is to combine the two internal arguments together and have the verb select the resulting constituent as its complement (1.1a). The arguments are described in terms of Williams' (1975) "small clauses" or taken to be connected by a silent preposition-like element (Kayne 1984; Pesetsky 1996; Harley 2002; Harley and Jung 2015). The alternative is to have the verb form a constituent with one of its internal arguments and then select the other one (1.1b). This option gives rise to VP-shells (Larson 1988) and analyses inspired by them (Kawakami 2018).



(a) Null P (adapted from Pesetsky 1996)

(b) VP-shells (adapted from Larson 1988)

Figure 1.1: The double object construction

Existing treatments of the double object construction generally fall into one of the two categories mentioned above, as there are only so many conceivable ways to form a binary-branching structure containing a verb and two arguments. That said, the abundance of recent literature on the topic indicates that this is far from a closed question. This and similar cases

2. Work concerning these structures also tends to assume and try to explain a connection between them and prepositional constructions, as in *John gave a book to Mary*. This too is a nontrivial analytical choice; see (Goldsmith 1980).

naturally lead to the follow-up question of this dissertation: when there is disagreement in the literature over a specific linguistic puzzle, how can the candidates be compared in terms of Chomsky's evaluation procedure? In other words, **how can one choose between different structures that could underlie the same linguistic construction based on quantitative considerations?**

In order to take on this question, one needs to capture precisely what makes the competing solutions different. This requires formalizing syntactic proposals at the algorithmic level, expressing them as a clearly defined set of building blocks and rules for putting them together. This dissertation adopts the formalism known as *minimalist grammars*, introduced by Stabler (1997). On the one hand, minimalist grammars were expressly designed as an implementation of Chomsky's Minimalist Program and offer a way to state analyses of syntactic phenomena in terms familiar to a linguist: lexical items defined by features and structure-building operations that combine them. On the other hand, they are explicit in spelling out assumptions about syntactic units and operations, and their formal properties – such as the complexity of string languages they generate and relation to other grammar formalisms – are relatively well understood. A semi-formal, example-driven description of minimalist grammars is given in Section 2.3.

1.3 Grammar optimization

In the literature, selecting a grammar to represent a language is usually framed as a *learning problem*. In terms of Gold (1967), a learner is a function that is presented with sentences from the target language and makes a guess about the language (in the form of one of its names, or grammars) after each example. The learner succeeds if after some point all its guesses are the same and a name of the correct language, regardless of the order in which examples are presented.

Within this paradigm, there is a substantial body of previous work concerned with learning grammars from unstructured strings; see e.g. an overview in (Clark 2017). These techniques are based around the notion of distributional similarity, not unlike that of Harris (1951); in short, words that occur in the same contexts are assigned the same syntactic category. Resulting algorithms can make linguistically plausible generalizations based on observable data. For example, Clark and Eyraud (2007) present an algorithm for learning a subclass of context-free grammars purely from positive data. It is capable of making certain generalizations – such as correctly generating sentences with both auxiliary fronting and a relative clause (*Is the man who is hungry ordering dinner?*) despite having only seen these phenomena separately in the training data. Yoshinaka (2011) generalizes this approach to a number of subclasses of multiple context-free grammars.

One question to ask at this point is whether the learner has to start with nothing but unstructured strings. Extralinguistic context of the input sentences as a source of information is central to the *semantic bootstrapping* theory of language acquisition laid out in (Pinker 1984, 1987). It works under the assumption that the learner can acquire meanings of many content words and construct a semantic representation of many input sentences (based on entities such as "thing", "true in the past", "predicate-argument relation") independently of learning grammatical rules. A more recent take on this idea (Siskind 1996; Stabler et al. 2003; Kobele et al. 2003) is that the learner can start the process of *grounding*, or mapping linguistic units to atoms of meaning, before learning the syntactic structure. Thus, it is plausible that the learner can identify relations between words of each sentence before knowing what lexical items or syntactic features are involved. This idea aligns with a different line of work, which focuses on producing grammars from annotated language data. Kobele et al. (2002) and Stabler et al. (2003) introduce a dependency-based approach: the learner is given a corpus of dependency structures – sentences segmented into morphemes and annotated with argument-structure and linear-order relations – and must generalize over separate instances of the same unit, converting the dependencies into a set of lexical items. Indurkhya (2019,

2020) proposes a method of inferring minimalist grammars from sentences annotated with semantic roles of arguments and agreement relations.

Approaches discussed so far aim to produce a description of a (typically infinite) language based on a finite sample drawn from that language. Using de la Higuera's (2010) terminology, such tasks can be referred to as *grammar induction* (when the primary goal is to obtain a grammar that explains the data) or *grammatical inference* (when a true target grammar is expected to exist, and the focus is on the quality of the learning process itself). In this dissertation, I take on a learning problem of a different kind. The goal is to compare and evaluate specific proposals of theoretical syntax, which is heavily reliant on highly abstract concepts. Two broad considerations are particularly illustrative in this respect:

- **Empty categories.** Phonetically empty lexical items (null DPs of various flavors, null heads such as complementizers, traces/unpronounced copies) are commonplace in the syntactic literature, even though they cannot be directly observed in the raw data. For example, a straightforward way to account for *that* being optional in sentences such as *Mary thinks that John is smart* vs. *Mary thinks John is smart* is to postulate a silent complementizer that introduces the subordinate clause *John is smart* in the latter case. While some recent methods of learning from annotated data can generate empty heads (Indurkhya 2019), works on learning from strings tend to prohibit silent elements altogether (Clark and Eyraud 2007; Yoshinaka 2011).

- **Morphology.** Many crucial generalizations require words to have internal structure. For example, the standard analysis of active constructions (*Mary praises John*) vs. passive constructions (*John is praised*) assumes they both use the same lexical verb *praise*, and the passive is derived by promoting the object (*John*) into the subject position. In order for this to work, the verb has to consist of at least two elements: the root, which is constant in the active and passive; and the suffix, which accounts for the contrast in the number and behaviour of arguments. A learning algorithm incapable

of manipulating units smaller than a word may be able to generate passive and active sentences correctly, but it will miss the generalization.

In order to zoom in on these and similar issues, I adopt a learning setting that emphasizes differences between weakly equivalent grammars. The input consists of a corpus of sentences and a minimalist grammar capable of generating them, defined over unsegmented words and in a maximally theory-neutral way. The learner's role is to refine the input grammar and produce a linguistically plausible description accounting for the original data. The focus is on capturing generalizations within the grammar rather than generalizing beyond the corpus; in fact, the output grammar may generate the same string language as the input. To distinguish this task from grammar induction or grammatical inference, I will use the term *grammar optimization*.

My proposal builds upon the notion of *lexical item decomposition* (Kobele 2018, to appear) to develop a systematic way to identify and eliminate redundancies across words. For example, the following observations – that *jumping*, *jumps*, *laughed*, and *walk* are intransitive verbs; that *jumping* and *jumps* share the same root; and that *jumps* and *laughed* are both finite – all translate into quantifiable similarities in their phonological and/or syntactic features. The atomic word *jumping* can then be split into two new lexical items: the root *jump*, which has the same syntactic properties as other intransitive roots, and the suffix *-ing*, which is shared by other gerund forms in the grammar. Units formed in this way may or may not have phonological content – in fact, empty lexical items play a crucial role in encoding distinctions between homophones and relations between syntactic categories. In essence, lexical decomposition factors out linguistic generalizations across the grammar and expresses them as new lexical items, transforming a grammar over unsegmented words into one of morphemes.

With this general strategy in mind, I define an evaluation metric for minimalist grammars and a set of operations over (sets of) lexical items based around the notion of lexical decomposition. These results are then used to design a procedure for grammar optimization.

As proof of concept, I also develop a Python implementation of this learning procedure and use it to perform a series of experiments optimizing descriptions of syntactic phenomena. This work focuses on English as the best-studied language with the largest available body of syntactic literature. That said, applying lexical decomposition to a language with richer morphology, such as Turkish or Swahili, may be an interesting task for the future. All materials used in the experiments, including the code for the optimization algorithm and input datasets, can be found online at https://github.com/mermolaeva/mg-optimizer.

The immediate goal of this work is to offer an additional tool to linguists. However, its higher-level intent is to capture a more general insight about the way syntactic analyses are structured and developed. From a broader perspective, I aim to demonstrate how the mainstream study of natural language syntax might benefit from a formally explicit approach, and ultimately help build a stronger connection between theoretical linguistics and the mathematical theory of formal languages.

1.4 Dissertation outline

Chapter 2: Background provides necessary mathematical preliminaries and an introduction to minimalist grammars, in order to keep the dissertation self-contained. An important caveat of the formal approach adopted here is that any results will be contingent on the selected formalism and its (by necessity very specific) assumptions about syntax. Therefore, I also briefly discuss how faithful the machinery of minimalist grammars is to theoretical proposals, and how it can be extended to better accommodate them.

Chapter 3: Comparing proposals addresses the problem of quantifying what makes a grammar good. It discusses the Minimum Description Length principle (Rissanen 1978) and the way it can be used to obtain an evaluation measure for minimalist grammars. It also provides a case study of a concrete language phenomenon, which shows how this

evaluation measure takes into account specific arguments from theoretical literature and reflects predictions of various competing analyses.

Chapter 4: Deconstructing syntactic generalizations introduces the concept of lexical item decomposition as the driving force behind syntactic analysis and unpacks this intuitive idea into a toolkit of elementary operations over (sets of) lexical items. This is followed up with a fully worked out sample transformation of a small grammar to demonstrate how complex generalizations can arise through repeated application of small, easily interpretable steps.

Chapter 5: Towards a learning algorithm develops an optimization procedure based on the evaluation measure and operations outlined in previous chapters. It introduces additional machinery required to apply lexical decomposition in a systematic way, form and evaluate hypotheses, and navigate the search space of grammars to converge to an optimal solution.

Chapter 6: Experiments reports the results of applying the optimization procedure to various datasets. It also examines different quantitative metrics and the effects they have on the behaviour of the learner. The experiments cover a wide range of syntactic phenomena including passives, complement clauses, and raising constructions; l-selection of prepositional phrases; and allomorphy of verb stems, and help to evaluate the algorithm's ability to converge on a concise, linguistically motivated description of (a nontrivial fragment of) the English grammar.

Chapter 7: Conclusions takes a step back to summarize the results and outline a number of directions for future work.

Chapter 2

Background

2.1 Mathematical preliminaries

2.1.1 Tuples, sets, relations, and functions

A *tuple* is defined as an ordered collection of objects. A tuple of two elements is called a *pair*, and a tuple of three elements a *triple*. Tuples are written in angle brackets. For any tuple $T = \langle t_1, ..., t_n \rangle$ of n elements, for $i, j \in [1, n]$, $T[i]$ denotes t_i, the i-th *component* of T, and $T[i, j]$ denotes $\langle t_i, ..., t_j \rangle$, the tuple containing the i-th to j-th components of T. The number i is called the *index* of t_i.

A *set* is defined as an unordered collection of distinct objects. Elements of a set are written in curly braces. Let $A = \{a_1, ..., a_n\}$ denote a set. Its *size*, $|A| = n$, is the number of elements in A. The set of all subsets of A, $\mathcal{P}(A)$, is called its *power set*. The *empty set* of size 0 is denoted $\varnothing$. $\mathbb{N}$ is the set of natural numbers, including 0. An unordered collection that allows multiple instances of objects is called a *multiset*.

Given two sets A and B, their *cross-product* $A \times B$ is defined as the set of all pairs $\langle a, b \rangle$ such that $a \in A$ and $b \in B$. Generalizing the notion of cross-product, the *finite product* of sets $A_1, A_2, ..., A_n$ is the set of all tuples of length n such that the i-th element of each

tuple is chosen from A_i, for $i \in [1, n]$: $A_1 \times A_2 \times ... \times A_n = \{\langle a_1, a_2, ..., a_n \rangle : a_1 \in A_1, a_2 \in A_2, ..., a_n \in A_n\}$. The rest of usual set operations are defined below:

$$A \cup B = \{x : x \in A \text{ or } x \in B\} \qquad (\textit{union})$$

$$A \cap B = \{x : x \in A \text{ and } x \in B\} \qquad (\textit{intersection})$$

$$A - B = \{x : x \in A \text{ and } x \notin B\} \qquad (\textit{difference})$$

For any set or tuple X, $X[x_1 \mapsto y_1, ..., x_k \mapsto y_k]$ denotes the structure (set or tuple) that is identical to X except for each occurrence of x_i which is replaced with y_i, for $i \in [1, k]$.

A *relation* on sets A and B is a subset of $A \times B$. A (partial) *function* f from A (its domain) to B (its codomain), written as $f : A \to B$, is a relation $f \subseteq A \times B$ such that for every $\langle a, b \rangle \in f$, for any b' such that $\langle a, b' \rangle \in f$, $b' = b$.

A relation R on a set A and itself is called a *partial order* over A if it has the following properties:

for any $x \in A, \langle x, x \rangle \in R$ (*reflexivity*)

for any $x, y \in A$, if $\langle x, y \rangle \in R$ and $\langle y, x \rangle \in R$, then $x = y$ (*antisymmetry*)

for any $x, y, z \in A$, if $\langle x, y \rangle \in R$ and $\langle y, z \rangle \in R$, then $\langle x, z \rangle \in R$ (*transitivity*)

For any $x, y \in A$, we say that x and y are *comparable* in X if $\langle x, y \rangle \in R$ or $\langle y, x \rangle \in R$.

An n-ary *operation* over a set A is defined as a function from $\underbrace{A \times ... \times A}_{n \text{ times}}$ to A. The *closure* of A under an operation f is the smallest set containing A such that f returns a member of the set when applied to members of the set.

2.1.2 Strings and languages

An *alphabet* Σ is a finite set of symbols. A *string* over Σ is a finite sequence of elements of Σ. Similar to the size of a set, the *length* of a string w is the numbers of symbols it contains, written as $|w|$. For any two strings u and v, $u \cdot v$ or uv denotes their *concatenation*. If $w = uv$, then u is a *prefix* of w and v is a *suffix* of w.

For any $n \in \mathbb{N}$, Σ^n is the set of all strings over Σ of length n. Σ^* is the set of all finite strings of elements of Σ, including the unique empty string ϵ of length 0, whereas Σ^+ is the set of all non-empty strings over Σ. A *language* over Σ is a subset of Σ^*. A *grammar* is a finite set of rules describing how to form, or *generate*, strings in some (finite or infinite) language.

2.1.3 Graphs and trees

A *graph* is defined as a pair $\langle V, E \rangle$, where V is a set of *vertices* (or *nodes*) and E is a set of *edges* connecting pairs of vertices. In a *directed* graph, edges have orientations and are defined as ordered pairs such that $E \subseteq V \times V$. A *path* from v_1 to v_n in a directed graph $\langle V, E \rangle$ is a sequence of distinct vertices $v_1, v_2, ..., v_n$ such that $\langle v_i, v_{i+1} \rangle \in E$, for all $i \in [1, n-1]$. A *cycle* is a path from a vertex to itself. A *subgraph* $\langle V', E' \rangle$ of a graph $\langle V, E \rangle$ is a graph whose vertex set V' and edge set E' are subsets of V and E, respectively. A *multigraph* is permitted to have multiple edges connecting the same nodes; that is, E is defined as a multiset rather than a set. Edges that connect the same pair of nodes are called *parallel edges*.

An *unordered tree* is a directed graph $\langle N, P \rangle$ which has no cycles and for which the following hold:

- For any $\langle n_1, n_2 \rangle \in P$, n_1 is a *parent* of n_2, and n_2 is a *child* of n_1. Every node in N has at most one parent;
- Every node *dominates* itself and its children, and for any $n_1, n_2, n_3 \in N$, if n_1 dominates n_2 and n_2 dominates n_3, then n_1 dominates n_3. N contains a single *root* node which does not have a parent and which dominates all nodes in N.

A node n_1 *strictly dominates* n_2 if it dominates n_2 and $n_1 \neq n_2$. A node is called a *leaf* if it has no children, and an *internal* node otherwise. For any $n_1, n_2 \in N$, n_1 and n_2 are called *siblings* if they have the same parent.

An *ordered tree* is a triple $\langle N, P, R\rangle$ where $\langle N, P\rangle$ is an unordered tree and R is a partial order such that two nodes are comparable in R if and only if they are siblings. The *yield* of an ordered tree rooted in node t is defined as follows:

$$yield(t) = \begin{cases} t & \text{if } t \text{ is a leaf} \\ yield(c_1) \cdot ... \cdot yield(c_n) & \text{otherwise, where } c_1, ..., c_n \text{ are children of } t \\ & \text{and for } i, j \in [1, n],\ \langle c_i, c_j\rangle \in R \text{ iff } i \leqslant j \end{cases}$$

2.2 Context-free grammars

Context-free grammars, also called phrase-structure grammars (Chomsky 1956), are a grammar formalism developed for describing syntactic structure in natural language, which serves as the starting point of Chomsky's (1965) Standard Theory. A context-free grammar is defined by specifying the following components:

- N, a finite set of *nonterminal symbols*. By convention, $\texttt{S} \in N$ is the *start symbol*;
- Σ, a finite set of *terminal symbols* disjoint from N;
- R, a finite set of *(rewrite) rules*. Each member of R is a pair $\langle \alpha, \beta\rangle$ (usually written as $\alpha \rightarrow \beta$), where $\alpha \in N$ and β is a (potentially empty) string of terminal and nonterminal symbols.

Rules are applied by replacing the nonterminal symbol on the left-hand side with the sequence on the right-hand side. The derivation begins with the start symbol and proceeds by applying rules until no nonterminal symbols are left in the string.

For a concrete example, consider a CFG with $N = \{\texttt{S}, \texttt{DP}, \texttt{VP}, \texttt{D}, \texttt{N}, \texttt{AUX}, \texttt{VG}\}$, $\Sigma = \{$*this*, *boy*, *laughs*, *is*, *laughing*$\}$, and R as given in 2.1. CFGs are often represented simply as a list of rewrite rules, since N and Σ are recoverable from R.

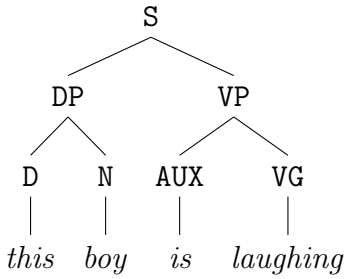
$$
\begin{aligned}
\texttt{S} &\rightarrow \texttt{DP VP} \\
\texttt{DP} &\rightarrow \texttt{D N} \\
\texttt{D} &\rightarrow \textit{this} \\
\texttt{N} &\rightarrow \textit{boy} \\
\texttt{VP} &\rightarrow \textit{laughs} \\
\texttt{VP} &\rightarrow \texttt{AUX VG} \\
\texttt{AUX} &\rightarrow \textit{is} \\
\texttt{VG} &\rightarrow \textit{laughing}
\end{aligned}
$$

Figure 2.1: A toy context-free grammar

A derivation of the string *this boy is laughing* and the associated phrase-structure tree, or *parse tree*, are shown in 2.2a and 2.2b respectively. In a phrase-structure tree for a context-free derivation, each internal node corresponds to the left-hand side of a rule, and its children to symbols on the rule's right-hand side.

$$
\begin{aligned}
&\texttt{S} \rightarrow \texttt{DP VP} \rightarrow \texttt{D N VP} \rightarrow \textit{this}\ \texttt{N VP} \rightarrow \\
&\textit{this boy}\ \texttt{VP} \rightarrow \textit{this boy}\ \texttt{AUX VG} \rightarrow \\
&\textit{this boy is}\ \texttt{VG} \rightarrow \textit{this boy is laughing}
\end{aligned}
$$

(a) Derivation



(b) Parse tree

Figure 2.2: Generation of *this boy is laughing* using 2.1

Context-free grammars have been shown by Shieber (1985) to be insufficiently powerful to describe some patterns found in natural language syntax. Nevertheless, they have useful connections to other grammar formalisms that will come in handy in Subsection 2.3.4.

2.3 Minimalist grammars

2.3.1 Lexical items, Merge, and Move

Minimalist grammars (MGs, Stabler 1997) provide a formal implementation of Minimalist syntax (Chomsky 1995, 2000), which is used throughout the dissertation. This section introduces the MG formalism and provides examples of derivations.

An MG specifies a finite set of lexical items and encodes their selectional properties in the form of *syntactic features*. A feature of the form `x` corresponds to a syntactic *category*, whereas `=x` and `x=` are selecting features which indicate that an expression is looking to merge (on the right or on the left, respectively[3]) with something of that category. Similarly, `-x` indicates the requirement to move, and `+x` means that the expression attracts a subexpression with that feature into its specifier position. In order to define an MG, one has to specify the following:

- *Base*, a finite set of syntactic *categories*. The set *Syn* of syntactic features is defined as the union of *Base* and the following sets:

$$\begin{aligned} Sel = &\{\texttt{=x} : \texttt{x} \in Base\} \cup && (\textit{right selectors}) \\ &\{\texttt{x=} : \texttt{x} \in Base\} && (\textit{left selectors}) \\ Lic = &\{\texttt{+x} : \texttt{x} \in Base\} \cup && (\textit{overt licensors}) \\ &\{\texttt{*x} : \texttt{x} \in Base\} && (\textit{covert licensors}) \\ Lee = &\{\texttt{-x} : \texttt{x} \in Base\} && (\textit{licensees}) \end{aligned}$$

  Each syntactic feature is then characterized by its *name* (drawn from *Base*) and *type* (category, right/left selector, overt/covert licensor, or licensee). Selectors and licensors together are called *attractors*, and categories and licensees are called *attractees*;

3. The choice to distinguish between left and right selection puts linear order under lexical control. One alternative, commonly adopted in the literature on MGs, is to have the first dependent of a head merge on the right, and all subsequent dependents on the left – a version of the Linear Correspondence Axiom (Kayne 1994).

- Σ, a finite alphabet of phonological segments;

- *Lex*, a lexicon, or finite set of *lexical items*. Each lexical item (LI) is a pair $\langle s, \delta \rangle$ (written as $s :: \delta$), where $s \in \Sigma^*$ is a (phonological) *string component* and $\delta \in Syn^*$ is a list of syntactic features, or *feature bundle*. We will sometimes refer to specific lexical items by their string components, where it does not lead to ambiguity.

MGs are commonly defined by simply stating a lexicon, which also implicitly fixes a set of categories and an alphabet of segments. Because of this, and for the sake of convenience, we will use the terms "grammar" and "lexicon" interchangeably when referring to MGs. An example grammar of five lexical items is given below:

this :: `=n d -k`
boy :: `n`
is :: `=g +k t`
laughing :: `=d g`
laughs :: `=d +k t`

Figure 2.3: A toy MG

Syntactic *expressions* generated by an MG are binary trees whose terminal nodes are labeled with LIs (which themselves are referred to as *atomic expressions*). The first feature of each LI is *syntactically active*, i.e. accessible to structure building operations. These operations, **merge** and **move**, consume matching attractors and attractees to generate complex expressions from *Lex*. The set of expressions *Exp* is defined as the closure of *Lex* under **merge** and **move**.

- **merge** : $(Exp \times Exp) \rightarrow Exp$ is a binary function that targets selectors and categories and combines two syntactic expressions into a new one. The dependent is merged on the left if the selector is of the form `x=`, and on the right if it is of the form `=x`.

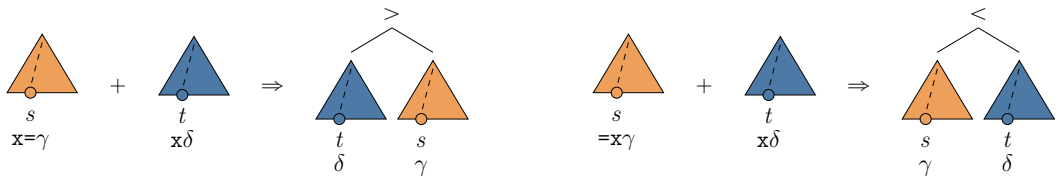


Figure 2.4: Left and right **merge**

- **move** : $Exp \times Exp$ is a relation that matches a licensor with a licensee within the same expression. Overt licensors (+x) cause the moving subtree to become a (left) sister of the head, leaving behind an empty node[4] without a string component or syntactic features. Covert **move** (*x) leaves the string component behind.[5]

Figure 2.5: Overt and covert **move**

While there are many ways to limit the number of features which may be syntactically active at any given time, a simple one with desirable computational properties stipulates that only one feature of each name may be the first feature of any feature bundle in an expression. In particular, this means that the number of movable subtrees in any expression is limited by the size of *Base*. This restriction is known as the Shortest Move Constraint, or SMC. With the SMC in place, **move** becomes a function.

A single lexical item (atomic expression) is considered its own *head*. For complex structures formed by **merge** or **move**, the expression with the attractor becomes the head of the new expression; and the one with the attractee becomes its *dependent*. We label the parent

4. We indicate a moved string t as $t\!\!\!/$. This is a notational convenience; formally, the empty node contains ϵ, the empty string.

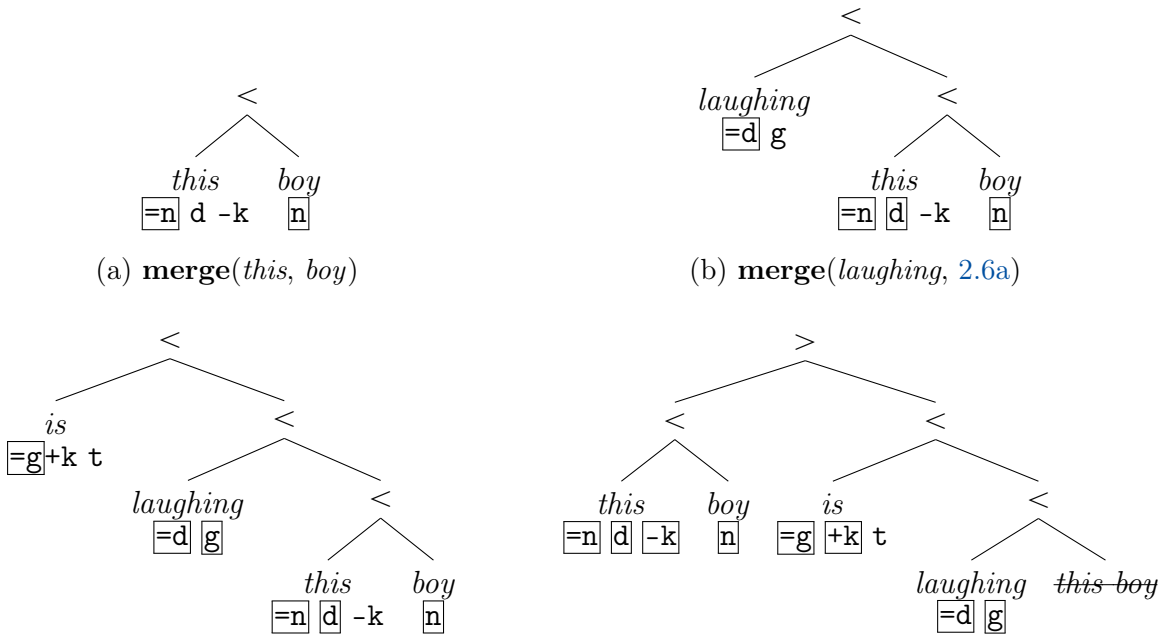
5. This version of covert movement, which displaces syntactic features but leaves the string component in its base position, is in line with (Stabler 1997). It fixes the position of a sub-expression once it has been covertly moved, rendering its string component inaccessible to future instances of (overt) **move**. Though restricted, this implementation has been used in previous work on MGs; see e.g. (Torr and Stabler 2016) and is sufficient for our purposes.

node with $<$ if the head is on the left or $>$ if the head is on the right. The dependent introduced by the first attractor of an LI is its *complement*, and all subsequent dependents are *specifiers*. Matched features are *checked*, or deleted, making the next feature in the bundle accessible for syntactic operations. Checked features are no longer visible to syntax. We will sometimes keep them in representations for clarity, in which case they will be marked as $\boxed{\texttt{x}}$.

An expression with no unchecked features except for some category `x` on its head is called a *complete expression* of that category. We will be primarily concerned with complete expressions of category `t` (for Tense) or `c` (for Complementizer) and their yields (*sentences*). By the definition of **merge**, a lexical item can only check its attractors as long as it is the head of the expression; and by the definition of **move**, it can only check its licensees after being selected and having its category checked by another lexical item. Therefore, in order to arrive at a complete expression of any category, each lexical item in the expression must have a feature bundle of the form $(Sel \cup Lic)^*\ Base\ (Lee)^*$.

The lexicon in 2.3 generates, among others, the four expressions in 2.6. In 2.6a, **merge** applies to *this* and *boy*, whose feature bundles start with the matching features `=n` and `n`, respectively. Both `=n` and `n` are deleted. In 2.6b, **merge** once again targets two expressions: *laughing*'s feature bundle starts with `=d`, and 2.6a has `d` as its first feature. Another **merge** step (2.6c) checks the `=g` and `g` features, combining *is* with 2.6b.

In 2.6d, the matching features are `+k` on *is* and `-k` on *this*. The DP is **move**d into the specifier position of *is*, which becomes the head of the new expression.



(a) **merge**(*this*, *boy*)

(b) **merge**(*laughing*, 2.6a)

(c) **merge**(*is*, 2.6b)

(d) **move**(2.6c)

Figure 2.6: Derivation of *this boy is laughing* using 2.3

We have arrived at a complete expression of category `t`, yielding the sentence *this boy is laughing*. Another sentence, *this boy laughs*, can be produced by applying **merge** to *laughs* and 2.6d, and then **move** to the resulting structure. These are the only two sentences generated by this example lexicon.

At a glance, MGs appear rather different from mainstream Minimalist syntax in numerous ways – with respect to the feature calculus, implementation of movement, locality, and other issues. However, many of these seeming points of disagreement are a matter of convenience and can be tweaked without altering the crucial computational properties of the formalism, and much of the machinery employed by Minimalist syntax can be implemented as extensions of MGs or translated into the bare-bones formalism. For an in-depth discussion of how faithful MGs are to Minimalist syntax, see (Graf 2013, pp. 96–125). Here, I will briefly focus on one example: the hierarchy of syntactic categories.

In order to bridge the apparent gap, let us consider an explicit theory of feature structures compatible with Chomsky's framework, proposed by Adger (2010). Lexical items are defined

as sets of category features (T, V, N, D, ...) and morphosyntactic features (case, number, person, ...), each specified as bearing a value (drawn from some finite set) or being unvalued. Category features are ordered according to a number of universal Hierarchies of Projection (HoPs). The definition of a well-formed syntactic object specifies that the projecting head must bear a higher-valued category feature than the dependent; HoPs thus constrain the building of structure.

As an illustration, consider three lexical items: *the*, *many*, *men*. In Adger's system, there is a HoP in the nominal domain specifying N < Num < D. By assigning the categories D, Num, and N to lexical items *the*, *many*, and *men* respectively, the system ensures that *the many men* and *the men* are well-formed, while **many the men* is ruled out.

How would such a constraint translate into MGs? One option is to extend the formalism to include an explicit partial order relation over *Base*, the set of categories. However, an implicit ordering is already present in the bare-bones MGs. Consider the lexicon in 2.7.

the :: `=num d -k`
many :: `=n num`
ϵ :: `=n num`
men :: `n`

Figure 2.7: An MG ensuring the correct ordering of nominal projections

Since **merge** is feature-driven, the ordering of lexical items within expressions generated by this grammar is already constrained by their feature bundles: *the* can select an expression headed by *many*, but not vice versa. Finally, all we need to ensure the optionality of *many* is a phonologically empty lexical item, ϵ :: `=n num`.[6] It represents the idea that expressions of category `num` have a more limited distribution than those of category `n`; that is, any expression that selects a `num` can also select an `n`, but not the other way around.

6. Silent lexical items whose only contribution to the expression is changing its category are of special interest to our goal and will be discussed more extensively in Section 4.2.

Having no explicit machinery to encode HoPs significantly increases the size of a lexicon. This is an example of a trade-off: the conceptual simplicity of bare-bones MGs is preserved at the cost of a less elegant solution to a specific problem, with the understanding that the formalism can be extended, if necessary, to accommodate this additional machinery,[7] and that doing so would not change its core computational properties.

2.3.2 Complex words

The basic formalism outlined so far does not offer a way of recognizing structure within words, and the derivation in 2.6 treats each word as an inseparable unit. This is an oversimplification, which leads to considerable redundancy and missed generalizations. For instance, the grammar does not reflect the fact that *laughing* and *laughs* have a common root, or that *laughs* shares certain syntactic properties with *is*.

To remedy this, we need a way to combine multiple syntactic heads into complex units corresponding to multimorphemic words. This gives rise to a number of questions regarding the syntax-morphology interface and morphology proper:

1. Where in the sentence are the complex syntactic heads pronounced?
2. How is morphological information transmitted between words?
3. How are the complex heads mapped to strings?

With respect to (1), multiple options have been explored in the literature. *Head movement* creates a chain of heads that is pronounced in the highest head position (2.8a). *Lowering* or *affix hopping*, on the other hand, allows an affix to attach to the head of its complement, with the whole word being pronounced in the lower position (2.8b).

7. An extension adding a hierarchy of projections to MGs is implemented in (Fowlie 2013), which augments the formalism with a partial order over selectors to handle adjunction in a concise and explicit way.

(a) Head movement (b) Lowering

Figure 2.8: Building complex heads

Both processes are widely taken to be present in English. In particular, finite auxiliaries undergo head movement to T, while finite lexical verbs have affixes lowered onto them. Evidence of this is presented by Pollock (1989): finite auxiliaries precede sentential negation and undergo subject-auxiliary inversion in questions, whereas lexical verbs do not.

Unification of head movement and lowering is one of the defining features of Brody's (2000) Mirror Theory. This operation creates morphological dependencies between heads (lexical items), combining them into a *morphological word*. Each head is characterized as *weak* or *strong*. A complex morphological word is pronounced in the position of its highest strong head, or the lowest head if it does not contain any strong head. In a similar vein, Arregi and Pietraszko (2018) propose a generalized account of head movement and lowering as high and low spellouts of a single syntactic operation, *unified head movement*, supporting it with evidence of successive cyclic lowering as well as lowering feeding head movement. This proposal differs from Mirror Theory in that the default position of a complex word, in the absence of strong heads, is the *highest* head position. Among other things, this immediately offers an analysis for the English data. Tense, auxiliary *be*, and auxiliary *have* are weak heads, whereas lexical verbs are strong. As a result, finite *have* and *be* are pronounced in the highest position, since they don't contain any strong heads, but finite lexical verbs are pronounced in the lowest position.

Some approaches to complex heads have been adapted for the MG formalism. Stabler (2001) incorporates both head movement and lowering into MGs as subtypes of selector features. This formalization renders the material in a lowered complex head inaccessible

for any future head movement or lowering. Brody's framework was adapted into minimalist grammars by Kobele (2002), and was proven not to affect the weak generative capacity of the formalism. Arregi and Pietraszko's (2018) proposal is similarly implemented in (Kobele to appear).

Question (2) deals with local and long-distance flow of information between words and its morphological manifestation. Chomsky's Minimalist program (2000; 2001) considers these phenomena an effect of a general mechanism called Agree, which is driven by feature matching and forms relations between lexical items. The standard version of Agree takes place between a probe and a goal with matching features, such that the probe c-commands the goal and there is no other eligible goal that is closer to the probe. To establish an Agree relation, the probe look downwards into its domain (sister), and the goal transmits feature values upwards to the probe. In Adger's (2010) system, this is made explicit as a standalone operation which targets morphosyntactic features and establishes dependencies within existing structure – in contrast to Merge and Move, which operate on category features and build new structure.

A framework for agreement compatible with MGs is outlined in (Ermolaeva 2018; Ermolaeva and Kobele 2019). This line of work utilizes a more refined definition of lexical items: instead of a string component, each LI is associated with a set of morphological features and values, which encode information needed to realize the head as a string. Morphological features are then valued in the course of the derivation, using dependencies formed by **merge** and **move**. For instance, rather than the string *boy*, the corresponding LI would be represented by a set containing, among others, the features `number`:*singular*, `person`:*3*, and `case`:*unvalued*, whereas the lexical item *be* would carry `number`:*unvalued*, `person`:*unvalued*, and `case`:*nominative*. Their respective `-k` and `+k` features would be marked as allowing transmission of feature values along the **move** dependency. Like many other modifications of MGs, operating over sets of morphological features provides a succinct way of formulating

generalizations but does not change the core properties of the formalism.[8] More generally, as shown by Graf (2013), restrictions on LI compatibility can be added to MGs as long as they are definable in monadic second-order logic. In this case, the constraints can be built into standard MGs by refining syntactic categories. These results are used directly, for instance, in (Laszakovits 2018) to implement dependent case in MGs.

Finally, (3) falls outside the domain of syntax, or even the syntax-morphology interface, and into morphology proper. While this issue is largely outside the scope of this dissertation, it represents another decision which must be made to ensure that the syntax formalism we are using is capable of supporting complex words. Generally speaking, this requires an MG-compatible theory of morphology, understood as a function mapping words (simple or complex heads output by an MG) to phonological strings. This function may be as simple as string concatenation or as involved as a faithful formalization of a morphological framework proposed in the literature.[9]

The various options available with respect to each of the three questions are meaningful and worth exploring in the context of grammar optimization. That said, just as with syntax proper, one needs to strike a balance between faithfulness and conceptual simplicity when making additions to the formalism. In order to focus on the issues outlined in Section 1.3,

8. For example, subject-verb agreement in English can be expressed, in a straightforward but cumbersome fashion, as a bare-bones MGs over immutable strings:

I :: d -k_{1SG}	*am* :: =g +k_{1SG} t
this :: =n_{3SG} d -k_{3SG}	*is* :: =g +k_{3SG} t
these :: =n_{3PL} d -k_{3PL}	*are* :: =g +k_{3PL} t
boy :: n_{3SG}	*walking* :: =d g
boys :: n_{3PL}	

Figure 2.9: Subject-verb agreement in an MG

This lexicon enforces agreement, as only lexical items with compatible string components may combine via **merge** or **move**. This is made possible by introducing a separate syntactic feature for each configuration of morphological properties resulting in a distinct morphological form.

9. For example, Ermolaeva and Edmiston (2018) propose a formalization of Distributed Morphology (Halle and Marantz 1993) that is explicitly designed to operate over expressions generated by chain-based minimalist grammars.

and keeping in mind English as the primary case study, I make the following simplifying assumptions:

- All complex heads are formed by head movement;
- Lexical items carry string components rather than sets of morphological features;
- Morphology is concatenative, and all affixation is suffixation.

Following Stabler (2001), this is implemented as an additional subtype of selector features, `=>x`, which triggers right **merge** accompanied by head movement. We will call these features *morphological selectors*. This version of head movement is defined in terms of head-complement relations, which means that the new type is restricted to the first feature in the bundle. More precisely, we add the set $\{\texttt{=>x} : \texttt{x} \in \mathit{Base}\}$ to *Sel* and define a new operation targeting these attractors:

- **merge** with head movement is triggered by selectors of the form `=>x`. It proceeds as right **merge** and concatenates the string component of the head of the complement with that of the resulting expression.

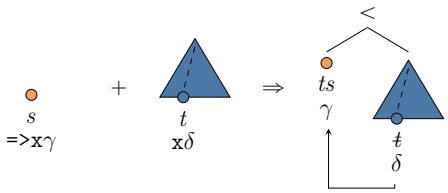


Figure 2.10: **merge** with head movement

We will refer to lexical items bearing these selector features as *affixes* and write their string components starting with a hyphen, following a common notational convention.

With this modification, our toy lexicon can be updated to reflect structure within inflected verbs. Instead of two atomic words, this grammar has a single lexical item representing the root, *laugh*, that can be selected by either of two affixes, *-ing* or *-s*.

this :: `=n d -k`

boy :: `n`

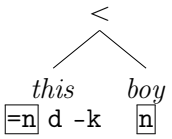
is :: `=g +k t`

laugh :: `=d v`

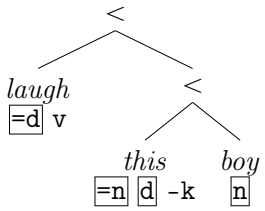
-ing :: `=>v g`

-s :: `=>v +k t`

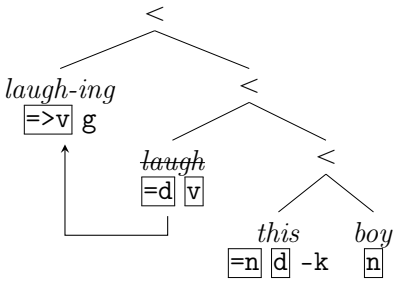
Figure 2.11: An MG with head movement



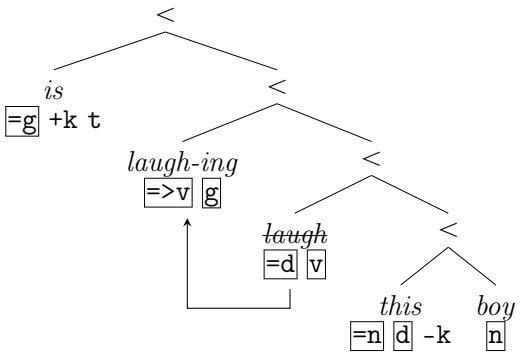
(a) **merge**(*this*, *boy*)



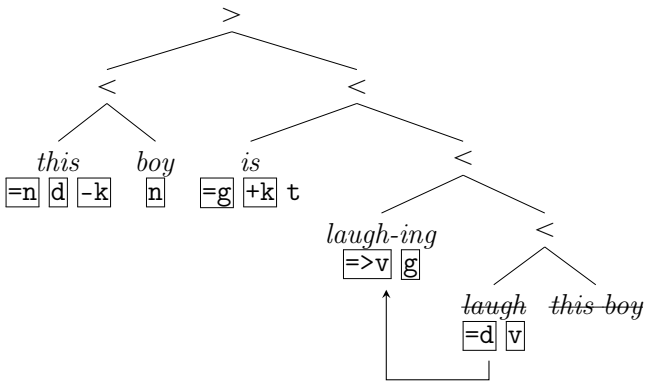
(b) **merge**(*laugh*, 2.12a)



(c) **merge**(*-ing*, 2.12b)



(d) **merge**(*is*, 2.12c)



(e) **move**(2.12d)

Figure 2.12: Derivation of *this boy is laugh-ing* using 2.11

Let us compare the derivation in 2.6 with its counterpart using the updated lexicon (2.12). The first two steps, 2.12a and 2.12b proceed in the same way. Next, we **merge** in *-ing*. Its selector feature, `=>v`, triggers head movement, concatenating *laugh* and *-ing* together (2.12c). The two remaining steps, **merge** and **move**, proceed as normal, resulting in the complete expression in 2.12e and yielding *this boy is laugh-ing*.

2.3.3 Representing minimalist grammars and expressions

So far, we have been using phrase-structure trees, or *derived trees* (2.12), to visualize syntactic expressions generated by MGs. Derived trees convey the linear order of lexical items; for a complete expression of category `t`, a sentence can be obtained from such a tree by computing its yield. On the other hand, a *derivation tree* (2.13) is a compact record of how the expression was generated. In a derivation tree, each internal node corresponds to a step in the derivation, an instance of **merge** or **move**, and the order of its children reflects their role in that step: the head precedes its dependent regardless of their relative order in the derived structure.

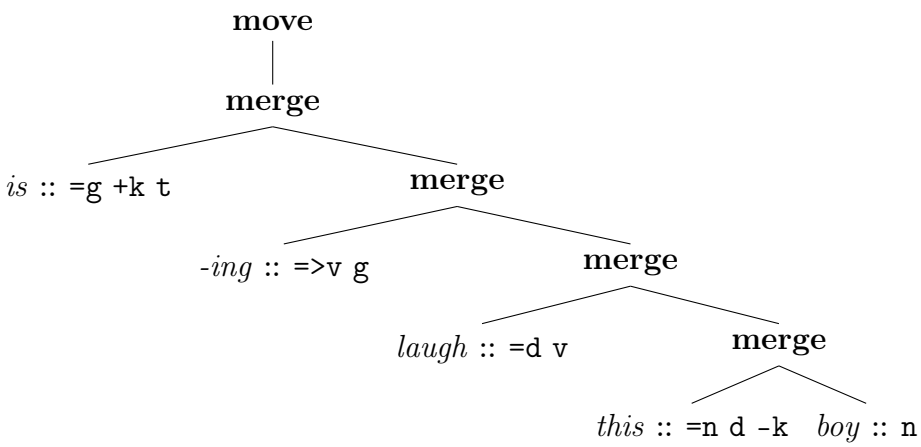


Figure 2.13: Derivation tree of *this boy is laugh-ing*

When it comes to representing an entire grammar, the default option is to list all lexical items, as in 2.11. As mentioned before, such a list contains all information required to define an MG. However, it does not provide a good overview of expressions generated by the grammar in question. While it works for very small toy examples, larger grammars with

dozens or hundreds of LIs can become difficult to read at a glance. A convenient alternative for visualizing head-complement relations within a set of lexical items is a directed multigraph whose vertices correspond to category features, and edges to lexical items. As an illustration, consider the graph of 2.11 below:

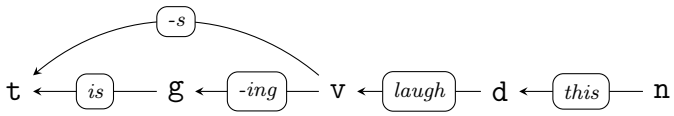


Figure 2.14: Head-complement relations within 2.11

This graph does not reflect all relations in the lexicon, since it ignores any **move** relations as well as any specifiers formed by **merge**. Lexical items without any selectors (such as *boy* :: `n`) don't contribute an edge to the graph. Instead, it focuses on a subset of relations which are relevant for morphologically complex words. Each path from `n` to `t` indicates a possible sequence of LIs along the clausal spine. Multiple paths between vertices indicate that there is more than one option available at that point in the derivation. For instance, there is an edge connecting `v` and `t`, as well as an alternative path between these categories. This reflects the fact that an expression of category `v` can be selected either by *-s* :: `=>v +k t` or by *-ing* :: `=>v g`, in the latter case producing a valid complement for *is* :: `=g +k t`.

2.3.4 Relation to CFGs

By definition, the two structure-building operations of MGs – **merge** and **move** – can only target subtrees whose heads bear an unchecked syntactic feature. Therefore, much of the derived structure is *syntactically inert*: once all features of a lexical item have been deleted, its position in the structure is fixed. The only elements that matter for syntax are those still capable of rearranging with respect to each other – namely, the head of the entire expression (via head movement) and any *movers*, or subtrees headed by lexical items with an unchecked licensee feature. With the SMC in place, the number of such subtrees in any given expression is finite, limited by the number of distinct licensee features in the grammar. Thus, a derived

tree can be flattened into a much more compact structure containing all information relevant for **merge** and **move** – a sequence of strings annotated with unchecked features.

This insight gives rise to the so-called *chain notation* for MGs (Stabler 2001; Stabler and Keenan 2003). In short, each expression sans movers is represented as an *initial chain* – a triple of strings corresponding to the head and material to its left and right, annotated with features of the head. Movers within the tree are represented by separate *non-initial chains*, the number of which cannot be greater than the size of *Base*.

$$\underbrace{(\textit{left},\ \textit{head},\ \textit{right}) : \texttt{features}}_{\text{Initial chain}},\ \underbrace{\textit{mover}_1 : \texttt{licensees},\ \textit{mover}_2 : \texttt{licensees},\ \ldots}_{\text{Non-initial chains}}$$

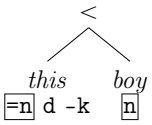
Figure 2.15: Schematic representation of a chain-based expression

Lexical items consist of only an initial chain, whose first and last components are empty strings, as shown in 2.15.

$$\begin{aligned}
\langle \epsilon, \textit{this}, \epsilon \rangle &:: \texttt{=n d -k} \\
\langle \epsilon, \textit{boy}, \epsilon \rangle &:: \texttt{n} \\
\langle \epsilon, \textit{is}, \epsilon \rangle &:: \texttt{=g +k t} \\
\langle \epsilon, \textit{laugh}, \epsilon \rangle &:: \texttt{=d v} \\
\langle \epsilon, \textit{-ing}, \epsilon \rangle &:: \texttt{=>v g} \\
\langle \epsilon, \textit{-s}, \epsilon \rangle &:: \texttt{=>v +k t}
\end{aligned}$$

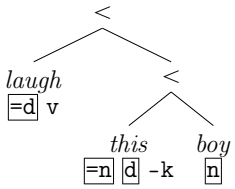
Figure 2.16: Chain-based version of 2.11

The structure-building operations are redefined in terms of string tuples. Informally, the outcome of **merge** depends on whether the dependent has reached its final position in the structure or is going to move later in the derivation. In the former case, its initial chain is concatenated together and attached to the leftmost (for left **merge**) or rightmost (for right **merge**) component of the initial chain. In the latter case, the dependent forms a non-initial chain ready to be targeted by **move**. Similarly, **move** comes in multiple varieties depending on whether the moving subtree has reached its surface position. A complete expression of category `x` consists of just an initial chain annotated with only the feature `x`.



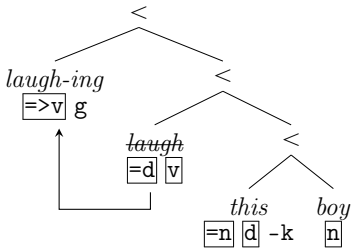
$\langle \epsilon, \textit{this}, \textit{boy} \rangle$: d -k

(a) **merge**(*this*, *boy*)



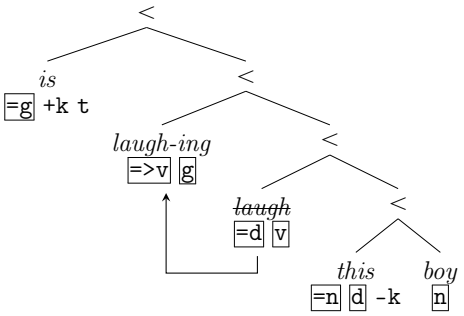
$\langle \epsilon, \textit{laugh}, \epsilon \rangle$: v, *this boy* : -k

(b) **merge**(*laugh*, 2.17a)



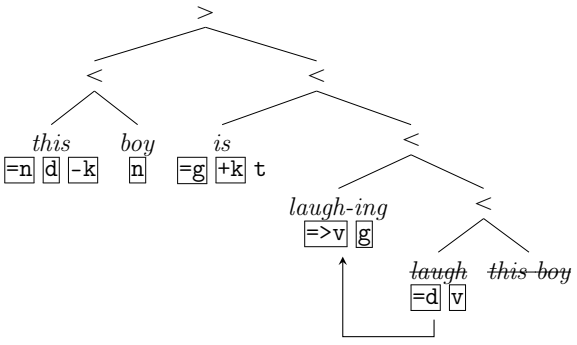
$\langle \epsilon, \textit{laugh-ing}, \epsilon \rangle$: g, *this boy* : -k

(c) **merge**(*-ing*, 2.17b)



$\langle \epsilon, \textit{is}, \textit{laugh-ing} \rangle$: +k t, *this boy* : -k

(d) **merge**(*is*, 2.17c)



$\langle \textit{this boy}, \textit{is}, \textit{laugh-ing} \rangle$: t

(e) **move**(2.17d)

Figure 2.17: Derivation steps of *this boy is laugh-ing* as chain-based expressions

The derivation of *this boy is laugh-ing*, shown before in 2.12, is repeated in 2.17, with each derivation step given as a derived tree and in chain notation side by side. In 2.17a, *this* and *boy* are **merge**d, and the string component of the latter is concatenated into the third component of the initial chain. Next, *laugh* is **merge**d with the resulting structure (2.17b). Since the dependent still carries a licensee feature (`-k`), it forms a non-initial chain *this boy* annotated with `-k`. The next two steps continue building up the initial chain, leaving the single non-initial chain unaffected. Finally, 2.17e **move**s *this boy* into the first component of the initial chain, arriving at a complete expression of category `t`.

Because chain notation is so compact, all intermediate steps in a derivation can be visualized as a single derivation tree by labeling each internal node with the chain-based expression corresponding to the step in question, as shown in 2.18.

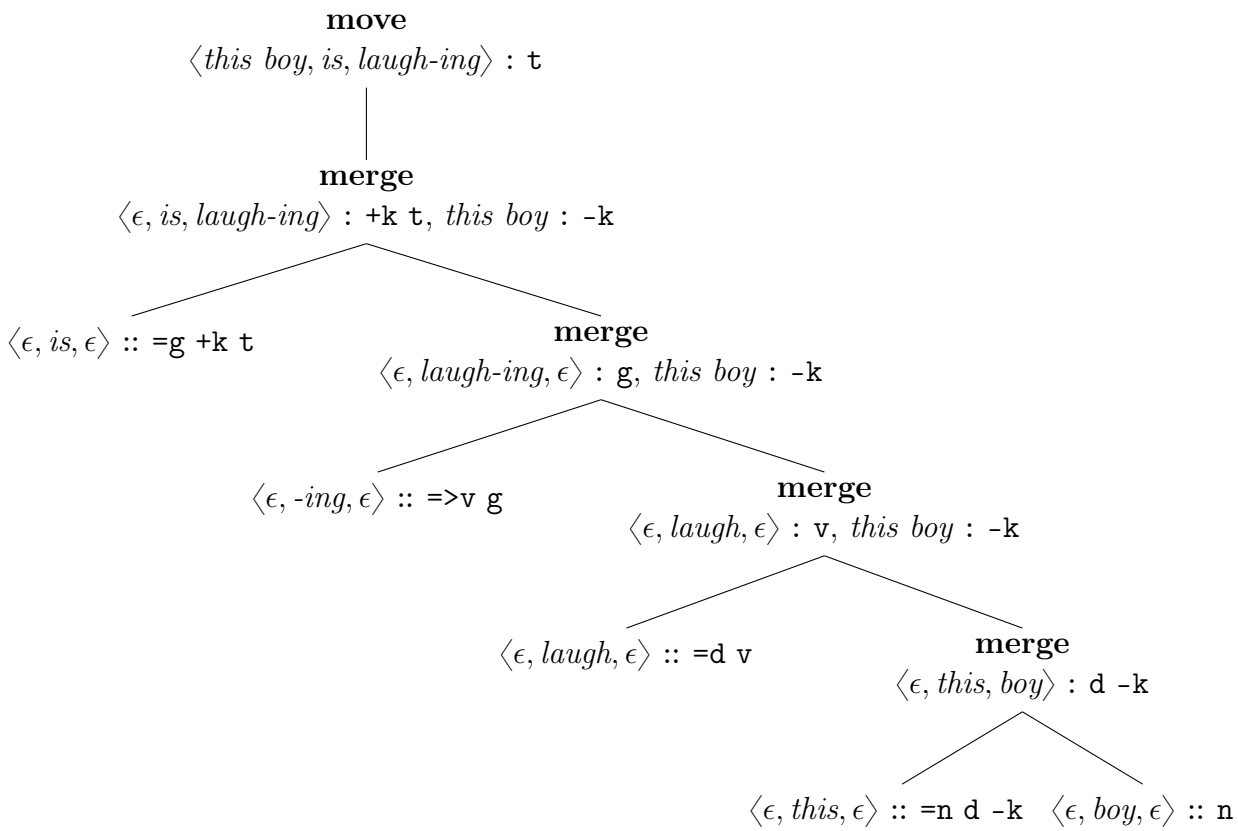


Figure 2.18: Chain-based derivation tree of *this boy is laugh-ing*

As mentioned before, derivation trees don't reflect displacement of leaves caused by **move** in the way derived trees do. For any MG, its derivation trees are parse trees of a CFG; a clear presentation of this result is given in (Hale and Stabler 2005). Intuitively, constructing this CFG can be thought of as pre-computing all possible derivation steps that can be performed by the MG. The central concept here is that of a *feature configuration*, which is obtained from a chain-based expression by omitting string components;[10] the SMC guarantees that the number of such configurations is finite. The set of feature configurations is obtained as the closure of the lexicon under **merge** and **move**. Informally, the conversion process is as follows:

- Each feature configuration (written in round brackets) becomes a nonterminal symbol;
- For each feature configuration formed by **merge** or **move**, there is a rule rewriting it as the operation's argument or arguments;
- For each LI, there is a rule rewriting its feature configuration as its string component;
- An additional rule rewrites the start symbol `S` as (`t`).

Derivation is then viewed as proceeding in the top-down manner of CFGs (starting with `t` and rewriting until lexical items in the leaves are reached), rather than the bottom-up manner characteristic of MGs. The CFG obtained from 2.11 is as follows:

$$
\begin{aligned}
\texttt{S} &\rightarrow (\texttt{t}) \\
(\texttt{t}) &\rightarrow (\texttt{+k t, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=g +k t})\ (\texttt{g, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=>v +k t})\ (\texttt{v, -k}) \\
(\texttt{g, -k}) &\rightarrow (\texttt{=>v g})\ (\texttt{v, -k}) \\
(\texttt{v, -k}) &\rightarrow (\texttt{=d v})\ (\texttt{d -k}) \\
(\texttt{d -k}) &\rightarrow (\texttt{=n d -k})\ (\texttt{n})
\end{aligned}
\qquad\qquad
\begin{aligned}
(\texttt{=n d -k}) &\rightarrow \textit{this} \\
(\texttt{n}) &\rightarrow \textit{boy} \\
(\texttt{=g +k t}) &\rightarrow \textit{is} \\
(\texttt{=d v}) &\rightarrow \textit{laugh} \\
(\texttt{=>v g}) &\rightarrow \textit{-ing} \\
(\texttt{=>v +k t}) &\rightarrow \textit{-s}
\end{aligned}
$$

Figure 2.19: CFG counterpart of 2.11

10. For covert movement, feature configurations should also indicate the non-initial chains whose string components have been left behind.

The method given in (Hale and Stabler 2005) is itself an adaptation of (Michaelis 1998), which shows how to convert an MG into an equivalent multiple context-free grammar (MCFG) generating the same language of sentences – yields of derived trees. MCFGs are a generalization of CFGs which operates on tuples instead of strings. Converting an MG into an equivalent MCFG is similar to constructing a CFG for yields, with a few differences. First, terminal rules rewrite feature bundles as triples of strings, corresponding to initial chains. Second, each non-terminal rule comes with a map describing how components of the argument tuples are rearranged and/or concatenated, in a way closely following chain-based **merge** and **move**.

Chapter 3

Evaluating proposals

3.1 The Minimum Description Length principle

Minimum Description Length (MDL, Rissanen 1978) is a principle for selecting a model to explain a dataset, which takes into account the simplicity of both the model itself and the explanation of the dataset it offers. In the MDL framework, the best grammar to describe a corpus is the one that minimizes the sum of the following:

- the length of the grammar, measured in bits;
- the length of the description assigned by the grammar to the corpus, measured in bits.

Within linguistics, MDL has been used, for example, for induction of phonological constraints (Rasin and Katzir 2016) and ordered rules (Rasin et al. 2018), morphological segmentation (Goldsmith 2001, 2006), and inferring syntactic categories given known morphological patterns (Hu et al. 2005).

To demonstrate this idea in action, let us examine a concrete example using CFGs. Consider a corpus of three strings over $\Sigma = \{this, boy, girl, laughs, jumps, and\}$:

this boy laughs;
this girl jumps;
this boy jumps and this girl laughs.

The three CFGs in 3.1 all generate these strings but assign different phrase-structure trees to them (3.2). The first one (3.1a) is too permissive and *overgenerates* by producing every non-empty string in Σ^*, including those that are not grammatical sentences of English, such as **laughs jumps girl and this this*. In linguistic terms, 3.1a assigns the same syntactic category to every word without regard to their distribution. The second grammar (3.1b) is too tight and *overfits* the corpus: it generates the three sentences above and nothing else. Finally, 3.1c strikes a balance by making a number of correct generalizations – for instance, that *boy* and *girl* have the same distribution and should be generated by the same nonterminal symbol. This grammar generates every sentence in the corpus, but also an infinite set of grammatical sentences absent from the corpus such as *this boy laughs and this girl jumps and this girl laughs*.

$$
\begin{aligned}
\texttt{S} &\rightarrow \texttt{X S} \\
\texttt{S} &\rightarrow \texttt{X} \\
\texttt{X} &\rightarrow \textit{this} \\
\texttt{X} &\rightarrow \textit{boy} \\
\texttt{X} &\rightarrow \textit{girl} \\
\texttt{X} &\rightarrow \textit{laughs} \\
\texttt{X} &\rightarrow \textit{jumps} \\
\texttt{X} &\rightarrow \textit{and}
\end{aligned}
$$

(a) Overgenerating

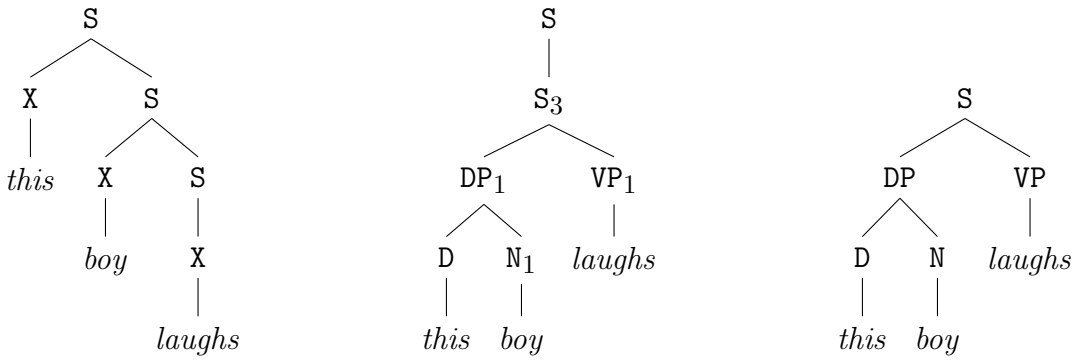
$$
\begin{aligned}
\texttt{S} &\rightarrow \texttt{S}_1\ \texttt{CONJ}\ \texttt{S}_2 \\
\texttt{S} &\rightarrow \texttt{S}_3 \\
\texttt{S} &\rightarrow \texttt{S}_4 \\
\texttt{S}_1 &\rightarrow \texttt{DP}_1\ \texttt{VP}_2 \\
\texttt{S}_2 &\rightarrow \texttt{DP}_2\ \texttt{VP}_1 \\
\texttt{S}_3 &\rightarrow \texttt{DP}_1\ \texttt{VP}_1 \\
\texttt{S}_4 &\rightarrow \texttt{DP}_2\ \texttt{VP}_2 \\
\texttt{DP}_1 &\rightarrow \texttt{D N}_1 \\
\texttt{DP}_2 &\rightarrow \texttt{D N}_2 \\
\texttt{D} &\rightarrow \textit{this} \\
\texttt{N}_1 &\rightarrow \textit{boy} \\
\texttt{N}_2 &\rightarrow \textit{girl} \\
\texttt{VP}_1 &\rightarrow \textit{laughs} \\
\texttt{VP}_2 &\rightarrow \textit{jumps} \\
\texttt{CONJ} &\rightarrow \textit{and}
\end{aligned}
$$

(b) Overfitting

$$
\begin{aligned}
\texttt{S} &\rightarrow \texttt{S CONJ S} \\
\texttt{S} &\rightarrow \texttt{DP VP} \\
\texttt{DP} &\rightarrow \texttt{D N} \\
\texttt{D} &\rightarrow \textit{this} \\
\texttt{N} &\rightarrow \textit{boy} \\
\texttt{N} &\rightarrow \textit{girl} \\
\texttt{VP} &\rightarrow \textit{laughs} \\
\texttt{VP} &\rightarrow \textit{jumps} \\
\texttt{CONJ} &\rightarrow \textit{and}
\end{aligned}
$$

(c) Balanced

Figure 3.1: Three context-free grammars



(a) Overgenerating (b) Overfitting (c) Balanced

Figure 3.2: Phrase-structure trees for *this boy laughs*

We will now see how this intuition translates into MDL values. For the sake of exposition, I adopt a straightforward encoding scheme after Katzir (2014). The first step is to convert each terminal symbol in N and each nonterminal symbol in Σ, along with an additional *delimiter* symbol, #, into a binary string. Then the number of bits needed to represent each symbol is

$$\lceil \log_2(|N| + |\Sigma| + 1) \rceil,$$

where $\lceil\ \rceil$ indicates rounding up to the nearest integer. It takes four bits to encode a symbol in 3.1a or 3.1c, while symbols of 3.1b require five bits each (3.3).

We can now use these binary representations to encode each grammar. Since context-free rewrite rules follow a very specific format (one nonterminal symbol on the left-hand side, a sequence of terminal and nonterminal symbols on the right-hand side), a grammar can be unambiguously represented by concatenating all symbols in each rule and concatenating all rules together, separated by delimiters, as shown in 3.4.

#	0000
S	0001
X	0010
this	0011
boy	0100
girl	0101
laughs	0110
jumps	0111
and	1000

(a) Overgenerating

#	00000
S	00001
S_1	00010
S_2	00011
S_3	00100
S_4	00101
DP_1	00110
DP_2	00111
D	01000
N_1	01001
N_2	01010
VP_1	01011
VP_2	01100
CONJ	01101
this	01110
boy	01111
girl	10000
laughs	10001
jumps	10010
and	10011

(b) Overfitting

#	0000
S	0001
DP	0010
D	0011
N	0100
VP	0101
CONJ	0110
this	0111
boy	1000
girl	1001
laughs	1010
jumps	1011
and	1100

(c) Balanced

Figure 3.3: Encoding tables for symbols

$$\underbrace{\texttt{S}}_{0001} \rightarrow \underbrace{\texttt{X}}_{0010}\ \underbrace{\texttt{S}}_{0001}\ \underbrace{\texttt{\#}}_{0000}\ \underbrace{\texttt{S}}_{0001} \rightarrow \underbrace{\texttt{X}}_{0010}\ \underbrace{\texttt{\#}}_{0000}\ \underbrace{\texttt{X}}_{0001} \rightarrow \underbrace{\textit{this}}_{0011}\ \underbrace{\texttt{\#}}_{0000} \ldots$$

Figure 3.4: Encoding of the overgenerating grammar (3.1a)

This step converts a grammar into a single binary string. Formalizing, the length of this string equals

$$\sum_{\langle \alpha, \beta \rangle \in R} \big(|\alpha| + |\beta| + 1\big) \times \big\lceil \log_2(|N| + |\Sigma| + 1) \big\rceil$$

and represents the size of the entire grammar in bits.

Our next step is to encode the data, which is done by using phrase-structure trees of sentences in the corpus. We start at the root (labeled with the start symbol, S) and traverse the tree in preorder – i.e. read the current node, then recursively traverse its children in the same way, from left to right. At each internal node, the number of possible choices equals the number of different rules whose left-hand side corresponds to the node's label. Formally, given the left-hand side α, the cost of encoding a rule in bits is

$$\left\lceil \log_2(|\{\beta : \langle \alpha, \beta \rangle \in R\}|) \right\rceil.$$

Using the overfitting grammar (3.1b) as an example, the cost of using the rule $\mathtt{S} \to S_3$ given the left-hand side S equals $\lceil \log_2 3 \rceil = 2$ bits, because there are 3 different rules whose left-hand side is S. If there is only one possible right-hand side, as with the rule $\mathtt{S}_3 \to \mathtt{DP}_1\ \mathtt{VP}_1$, the cost is 0 bits because there is no choice to make, and the corresponding encoding is ϵ, the empty string.

$\mathtt{S} \to \mathtt{X}\ \mathtt{S}$	0
$\mathtt{S} \to \mathtt{X}$	1
$\mathtt{X} \to$ *this*	000
$\mathtt{X} \to$ *boy*	001
$\mathtt{X} \to$ *girl*	010
$\mathtt{X} \to$ *laughs*	011
$\mathtt{X} \to$ *jumps*	100
$\mathtt{X} \to$ *and*	101

(a) Overgenerating

$\mathtt{S} \to \mathtt{S}_1\ \mathtt{CONJ}\ \mathtt{S}_2$	00
$\mathtt{S} \to \mathtt{S}_3$	01
$\mathtt{S} \to \mathtt{S}_4$	10
$\mathtt{S}_1 \to \mathtt{DP}_1\ \mathtt{VP}_2$	ϵ
$\mathtt{S}_2 \to \mathtt{DP}_2\ \mathtt{VP}_1$	ϵ
$\mathtt{S}_3 \to \mathtt{DP}_1\ \mathtt{VP}_1$	ϵ
$\mathtt{S}_4 \to \mathtt{DP}_2\ \mathtt{VP}_2$	ϵ
$\mathtt{DP}_1 \to \mathtt{D}\ \mathtt{N}_1$	ϵ
$\mathtt{DP}_2 \to \mathtt{D}\ \mathtt{N}_2$	ϵ
$\mathtt{D} \to$ *this*	ϵ
$\mathtt{N}_1 \to$ *boy*	ϵ
$\mathtt{N}_2 \to$ *girl*	ϵ
$\mathtt{VP}_1 \to$ *laughs*	ϵ
$\mathtt{VP}_2 \to$ *jumps*	ϵ
$\mathtt{CONJ} \to$ *and*	ϵ

(b) Overfitting

$\mathtt{S} \to \mathtt{S}\ \mathtt{CONJ}\ \mathtt{S}$	0
$\mathtt{S} \to \mathtt{DP}\ \mathtt{VP}$	1
$\mathtt{DP} \to \mathtt{D}\ \mathtt{N}$	ϵ
$\mathtt{D} \to$ *this*	ϵ
$\mathtt{N} \to$ *boy*	0
$\mathtt{N} \to$ *girl*	1
$\mathtt{VP} \to$ *laughs*	0
$\mathtt{VP} \to$ *jumps*	1
$\mathtt{CONJ} \to$ *and*	ϵ

(c) Balanced

Figure 3.5: Encoding tables for rules

In this way, we can now give binary string representations to all rules, as shown in 3.5. To encode a tree, we concatenate all rule encodings in the order in which the nodes are traversed (3.6).

$$\underbrace{\texttt{S} \to \texttt{X S}}_{0} \quad \underbrace{\texttt{X} \to \textit{this}}_{000} \quad \underbrace{\texttt{S} \to \texttt{X S}}_{0} \quad \underbrace{\texttt{X} \to \textit{boy}}_{001} \quad \underbrace{\texttt{S} \to \texttt{X}}_{1} \quad \underbrace{\texttt{X} \to \textit{laughs}}_{011}$$

(a) Overgenerating

$$\underbrace{\texttt{S} \to \texttt{S}_3}_{01} \quad \underbrace{\texttt{S}_3 \to \texttt{DP}_1\ \texttt{VP}_1}_{\epsilon} \quad \underbrace{\texttt{DP}_1 \to \texttt{D N}_1}_{\epsilon} \quad \underbrace{\texttt{D} \to \textit{this}}_{\epsilon} \quad \underbrace{\texttt{N}_1 \to \textit{boy}}_{\epsilon} \quad \underbrace{\texttt{VP}_1 \to \textit{laughs}}_{\epsilon}$$

(b) Overfitting

$$\underbrace{\texttt{S} \to \texttt{DP VP}}_{1} \quad \underbrace{\texttt{DP} \to \texttt{D N}}_{\epsilon} \quad \underbrace{\texttt{D} \to \textit{this}}_{\epsilon} \quad \underbrace{\texttt{N} \to \textit{boy}}_{0} \quad \underbrace{\texttt{VP} \to \textit{laughs}}_{0}$$

(c) Balanced

Figure 3.6: Encoding of *this boy laughs*

This explicit encoding scheme highlights the differences in how each grammar describes the data. Overall costs for the three grammars and data are given in Table 3.1. The overgenerating grammar (3.1a) is very short but requires a lengthy encoding for the corpus. The overfitting grammar (3.1b) makes describing the corpus extremely easy at the cost of a long encoding of the grammar itself.

	Grammar	Corpus	MDL
Overgenerating (3.1a)	100	52	152
Overfitting (3.1b)	265	6	271
Balanced (3.1c)	124	13	137

Table 3.1: Encoding costs for 3.1a–3.1c (bits)

The sum of the grammar and corpus encoding favors the balanced grammar (3.1c) – which aligns with a linguist's intuition of which of the three grammars is best.

3.2 Encoding minimalist grammars

A similar approach can be used to implement an MDL-based metric for MGs. Consider the following four sentences:

Mary laughs;
Mary laughed;
Mary jumps;
Mary jumped.

There are multiple (in fact, infinitely many) ways to construct a minimalist grammar accounting for this small corpus. Three of them are given in 3.7a, 3.7b, and 3.7c.

Mary :: `d -k`	*Mary* :: `d -k`	*Mary* :: `x -k`
laughs :: `=d +k t`	*laugh* :: `=d v`	*laugh* :: `=x x`
laughed :: `=d +k t`	*jump* :: `=d v`	*jump* :: `=x x`
jumps :: `=d +k t`	*-s* :: `=>v +k t`	*-s* :: `=>x +k t`
jumped :: `=d +k t`	*-ed* :: `=>v +k t`	*-ed* :: `=>x +k t`
(a)	(b)	(c)

Figure 3.7: Three minimalist grammars

The first two grammars, 3.7a and 3.7b, generate the four sentences above and no others. While they are are *weakly equivalent*, i.e. generate exactly the same set of strings, the structures they assign to these strings are different. In linguistic terms, the former treats each sentence as a single `t`P headed by an unsegmented verb (3.8a). The latter reanalyzes each finite verb form as a complex head formed by head movement. The lexical verb directly selects its argument and forms a `v`P, while the affix takes the `v`P as its complement and is responsible for the movement of the subject into its specifier position (3.8b). The third grammar, 3.7c, is also capable of generating inflected verbs in two derivation steps (3.8c). However, it conflates the category feature of lexical verbs with that of DPs, producing ungrammatical strings like **Mary-ed* and **Mary laugh-s* $(jump)^+$ (3.9).

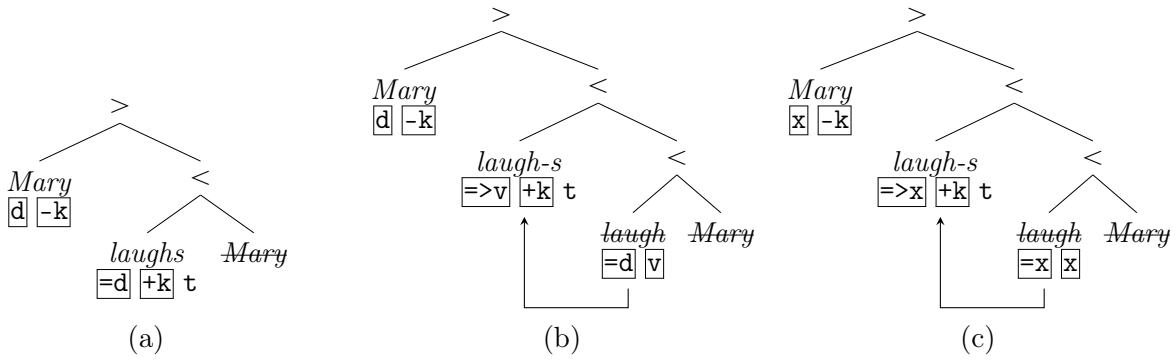


Figure 3.8: Structural differences between 3.7a, 3.7b, and 3.7c

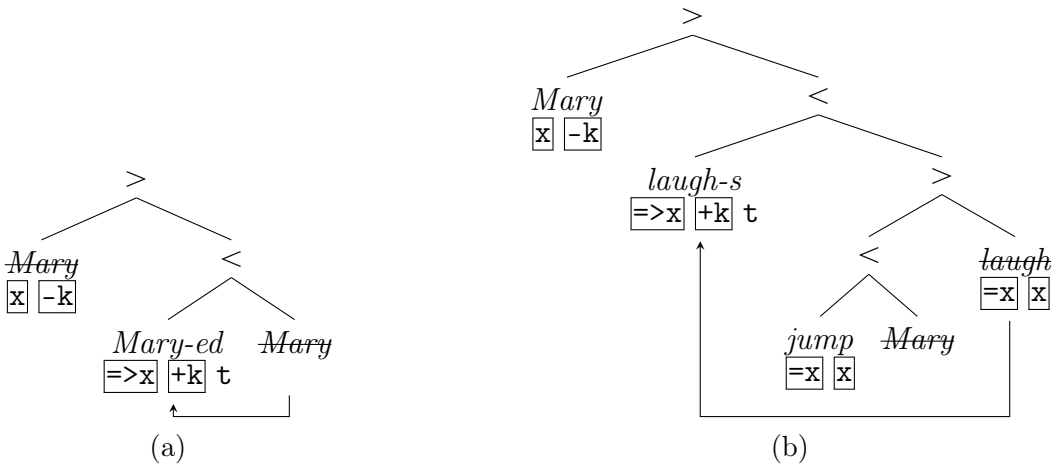


Figure 3.9: Overgeneration by 3.7c

To further help visualize the differences between these grammars, their graph representations are given in 3.10.

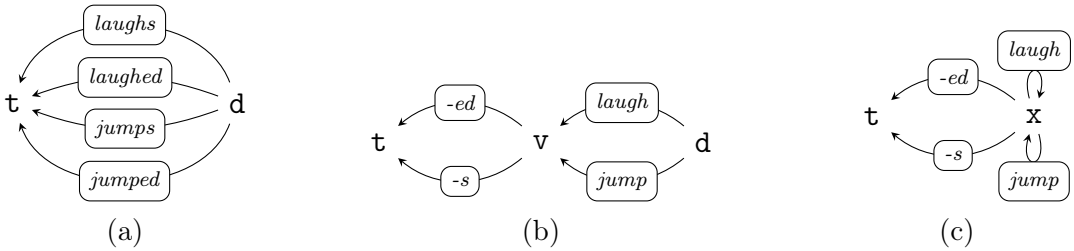


Figure 3.10: Graph representations of 3.7a, 3.7b, and 3.7c

For instance, *laughed* in 3.7a corresponds to one of the edges from `d` to `t` in the graph 3.10a. Its counterpart in 3.7b is a bimorphemic word, which translates into a pair of adjacent edges: *laugh* from `d` to `v` and *-ed* from `v` to `t` (3.10b). In 3.7c, lexical verbs correspond to loops.

Intuitively, 3.7b is an improvement over 3.7a. By recognizing internal structure within verbs, it captures the similarities within verbal paradigms (*laughs*, *laughed* vs. *jumps*, *jumped*) and across paradigms (*laughs*, *jumps* vs. *laughed*, *jumped*). On the other hand, 3.7a misses all these generalizations. For each new verbal paradigm encountered in the corpus (e.g. *walks*, *walked*), we would need to add two new lexical items to 3.7a, but only one to 3.7b. Finally, 3.7c is a subpar choice: it shares the desirable generalizations of 3.7b but also conflates a crucial distinction between two syntactic categories, leading to overgeneration.

What quantitative data can be used to back up this intuition? We can define an encoding scheme for MGs closely mirroring the one for context-free rules from Section 3.1. Since we are interested in the length of the encoding but don't need to calculate the binary string itself, we no longer round up to the nearest integer. Let $Types = \{$*category, right selector, left selector, morphological selector, overt licensor, covert licensor, licensee*$\}$ denote the set of syntactic feature types, and let Σ be the set of English letters. Then the size of a minimalist lexicon Lex over a set of categories $Base$ is given by

$$\underbrace{\sum_{s::\,\delta\,\in Lex} \big(|s| + 2 \times |\delta| + 1\big)}_{\text{total number of symbols}} \times \underbrace{\log_2(|\Sigma| + |\mathit{Types}| + |\mathit{Base}| + 1)}_{\text{cost of encoding per symbol}}.$$

Assuming that both Σ and $Types$ are fixed (with $|Types| = 7$ and $|\Sigma| = 26$[11]), this is a function of the number of LIs and the following three metrics:

- $|Base|$, the number of unique category features in Lex;
- $\sum_{syn} = \sum_{s::\,\delta\,\in Lex} \big(|\delta|\big)$, the total count of syntactic features in Lex;
- $\sum_{phon} = \sum_{s::\,\delta\,\in Lex} \big(|s|\big)$, the total length of all string components in Lex.

11. For simplicity, uppercase and lowercase letters are treated as the same symbol.

Regardless of the specific encoding scheme,[12] all three values above contribute to the size difference between grammars. The following table summarizes the differences between 3.7a, 3.7b, and 3.7c with respect to individual metrics, as well as grammar size:

	$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar (bits)
3.7a	3	14	28	317.78
3.7b	4	12	16	236.16
3.7c	3	12	16	234.43

Table 3.2: Grammar metrics

All three grammars have the same number of lexical items. However, splitting verbs into roots and affixes in 3.7b comes at the cost of an extra category feature. This pays off by eliminating redundant strings, which almost halves $\sum_{phon}$. Moreover, four instances of `+k` are collapsed into two, yielding a small reduction of $\sum_{syn}$. The differences would be much more noticeable with larger datasets, especially with respect to open-class words, since adding a new verb to 3.7a would have a higher cost (in both syntactic features and string components) compared to 3.7b.

It is also easy to see how a complexity measure based solely on grammar encoding would fail to penalize overgeneration. It would incorrectly favor 3.7c over 3.7b, given that it achieves the same reduction of $\sum_{phon}$ and $\sum_{syn}$ without increasing $|Base|$. Similar to the results observed with CFGs, the MDL component expected to rule out the overgenerating grammar is the corpus size given the grammar. In order to calculate it, for each MG we construct a CFG generating its derivation trees, as we did in Subsection 2.3.4, and then reuse the encoding scheme from Section 3.1. The CFGs are given in 3.11. Parse trees for *Mary laughs* as well as 3.11c's ungrammatical structures are shown in 3.12 and 3.13 respectively .

12. The solution used here is a naive one and serves to keep the example straightforward. The choice of an encoding scheme is a meaningful decision that can lead to different grammars being optimal for the same corpus.

$$
\begin{aligned}
\texttt{S} &\rightarrow (\texttt{t}) \\
(\texttt{t}) &\rightarrow (\texttt{+k t, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=d +k t})\ (\texttt{d -k}) \\
(\texttt{d -k}) &\rightarrow \textit{Mary} \\
(\texttt{=d +k t}) &\rightarrow \textit{laughs} \\
(\texttt{=d +k t}) &\rightarrow \textit{laughed} \\
(\texttt{=d +k t}) &\rightarrow \textit{jumps} \\
(\texttt{=d +k t}) &\rightarrow \textit{jumped}
\end{aligned}
$$

(a)

$$
\begin{aligned}
\texttt{S} &\rightarrow (\texttt{t}) \\
(\texttt{t}) &\rightarrow (\texttt{+k t, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=>v +k t})\ (\texttt{v, -k}) \\
(\texttt{v, -k}) &\rightarrow (\texttt{=d v})\ (\texttt{d -k}) \\
(\texttt{d -k}) &\rightarrow \textit{Mary} \\
(\texttt{=d v}) &\rightarrow \textit{laugh} \\
(\texttt{=d v}) &\rightarrow \textit{jump} \\
(\texttt{=>v +k t}) &\rightarrow \textit{-s} \\
(\texttt{=>v +k t}) &\rightarrow \textit{-ed}
\end{aligned}
$$

(b)

$$
\begin{aligned}
\texttt{S} &\rightarrow (\texttt{t}) \\
(\texttt{t}) &\rightarrow (\texttt{+k t, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=>x +k t})\ (\texttt{x, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=>x +k t})\ (\texttt{x -k}) \\
(\texttt{x, -k}) &\rightarrow (\texttt{=x x})\ (\texttt{x -k}) \\
(\texttt{x, -k}) &\rightarrow (\texttt{=x x})\ (\texttt{x, -k}) \\
(\texttt{x -k}) &\rightarrow \textit{Mary} \\
(\texttt{=x x}) &\rightarrow \textit{laugh} \\
(\texttt{=x x}) &\rightarrow \textit{jump} \\
(\texttt{=>x +k t}) &\rightarrow \textit{-s} \\
(\texttt{=>x +k t}) &\rightarrow \textit{-ed}
\end{aligned}
$$

(c)

Figure 3.11: CFG counterparts of 3.7a, 3.7b, and 3.7c

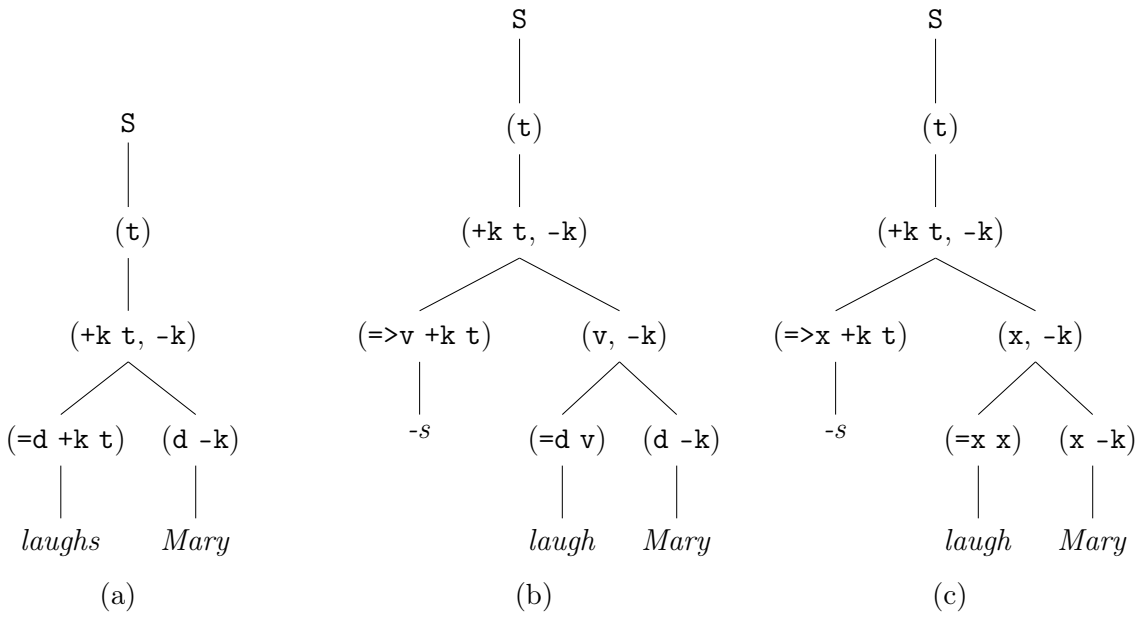


Figure 3.12: CFG parse trees: structural differences between 3.11a, 3.11b, and 3.11c

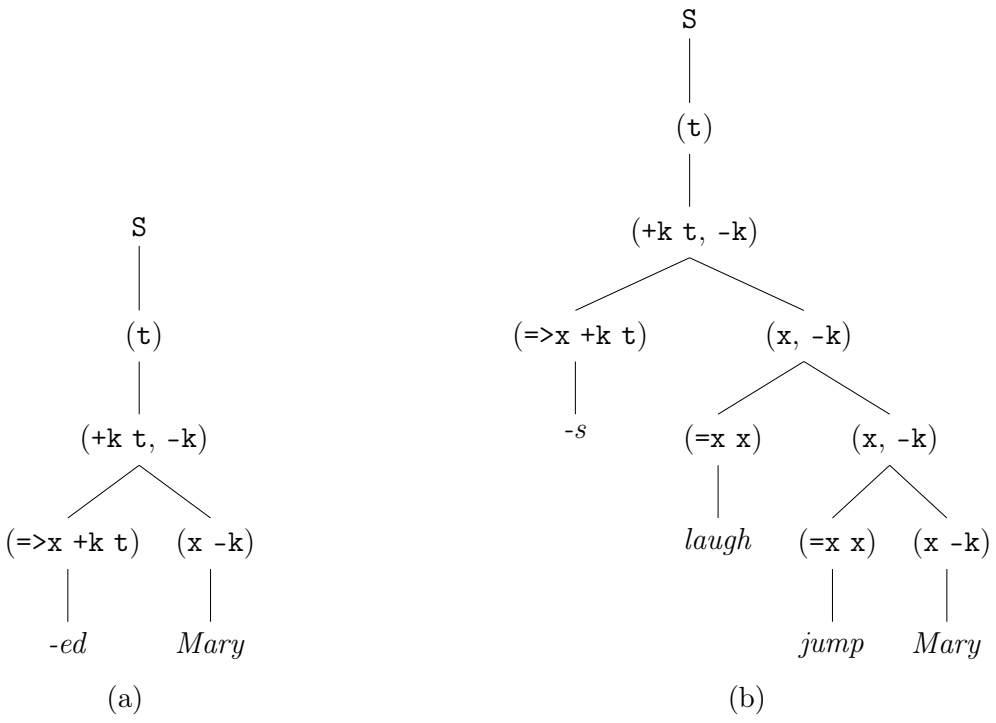


Figure 3.13: CFG parse trees: overgeneration by 3.11c

The cost of encoding the corpus given 3.11a is straightforward to calculate: there is only one choice with four options to be made in the derivation, namely rewriting (=d +k t) as

laughs, *laughed*, *jumps*, or *jumped*. In 3.11b this corresponds to two binary choices: rewriting (`=d v`) as *laugh* or *jump*, and (`=>x +k t`) as *-s* or *-ed*. Both cost 2 bits per sentence. The third grammar (3.11c), however, has two options for rewriting (`+k t`, `-k`) and two ways to expand (`x`, `-k`). These are the choices that make possible the ungrammatical strings in 3.13, but they also drive up the cost of encoding each grammatical sentence to 4 bits. This is summarized below:

	Grammar	Corpus	MDL
3.7a	317.78	8	325.78
3.7b	236.16	8	244.16
3.7c	234.43	16	250.43

Table 3.3: Encoding costs (bits)

Once we take the length of corpus encoding into account, the overgenerating grammar 3.7c is outperformed by the intuitively superior grammar 3.7b.

3.3 Double object construction revisited

We will now take a step up from toy examples towards more interesting applications of the technique introduced above and re-examine the double object construction in the light of MDL. As pointed out in Section 1.2, there are two groups of approaches to sentences like *John gave Mary a book*: those which postulate a small clause complement of *give*, and those which maintain that the double object construction is monoclausal. Enumerating and analyzing all known arguments from both sides in a comprehensive way falls outside of the scope of this dissertation. Instead, this section serves as proof of concept; in what follows, I convert a small sample of these arguments into the MG formalism and examine how the predictions of each analysis translate into higher or lower MDL values.

Let us focus on two facts regarding the English double object construction coming from two different sources. The first one is Harley and Jung (2015), who point out multiple parallels between double object structures with *give* and sentences with *have*. These are used to motivate an analysis where both *have* and *give* contain a possessive small clause headed by the abstract silent element PHAVE. One of there parallels is an animacy restriction. Both possessors in *have*-clauses (1a, 1c) and Goal arguments in *give*-clauses (1b, 1d) are required to be animate, as long as the possession is inalienable.

(1) a. John has a book.

b. Brenda gave John a book.

c. #The car has a flyer.

d. #The advertiser gave the car a flyer.

(Harley and Jung 2015, p. 704)

The second source is Kawakami (2018), who argues against the small-clause analysis, citing a number of discrepancies between the properties of known small clause constructions (e.g. *John considers Mary angry*) and those of *give*-clauses. One of the arguments supporting this stance comes from wh-movement and ambiguity. For sentences with *consider* (2a), both the matrix clause and the small clause can be modified by *why*, yielding two different interpretations. On the other hand, the double object construction behaves as monoclausal, allowing only one reading where *why* modifies the matrix clause (2b).

(2) a. Why did John consider Mary angry at Bill?

READING: asking the reason of considering

asking the reason of being angry

b. Why did John give Mary a book?

READING: asking the reason of giving

#asking the reason of having

(Kawakami 2018, pp. 220–221)

Which of these two arguments is stronger with respect to encoding costs? We start by translating each of them into an MG. Assuming a consensus on all issues other than the double object construction, the two grammars should share most of their LIs. Since this example involves wh-movement, we consider complete expressions of category `c` rather than `t`. The shared lexical items are given in 3.14a, and the additional LIs for *have* and *give* in the monoclausal and SC account are presented in 3.14b and 3.14c respectively. In accordance with the simplifying assumptions stated in Subsection 2.3.2, we ignore non-concatenative morphology and assume a separate set of morphological rules which realize *have-s* as *has* and *do-s* as *does*.

	consider :: =sc V	*angry* :: a
John :: d_a -k	-ϵ :: =>V +k d= v	ϵ :: =a d= sc
Mary :: d_a -k	-ϵ :: =>v x	*why* :: w -wh
the car :: d -k	*do* :: =x do	ϵ :: =sc w= sc
a flyer :: d -k	-ϵ :: =>x do	-ϵ :: =>t +wh c
ϵ :: =d_a +k d -k	-*s* :: =>do +k t	ϵ :: =t c

(a) Shared lexical items

		ϵ :: =d +k d_a= sc_{poss}
ϵ :: =d +k d_a= sc	ϵ :: =d +k d_a= sc	-ϵ :: =>sc_{poss} sc
have :: =sc v	*have* :: =sc v	*have* :: =sc_{poss} v
give :: =d +k d= V	*give* :: =sc V	*give* :: =sc_{poss} V
(b) Monoclausal *give*	(c) Uniform SC *give*	(d) Refined SC *give*

Figure 3.14: MG implementations of the double object construction

The simple solution in 3.14c views all small clauses as having the same syntactic category, `sc`. This validates Kawakami's (2018) objections to the small clause analysis based on multiple differences between small clauses selected by consider and arguments of give. However, Harley and Jung (2015, p. 718) point out a way to reconcile the two groups of phenomena, suggesting a typology of small clauses. Under this view, small clauses embedded under

consider (unlike those under *give*) include an additional projection, which explains different properties. Translating this idea into MGs yields the set of LIs given in 3.14d. Possessive small clauses ($\texttt{sc}_{\texttt{poss}}$) are selected by both *have* and *give*, and may merge with an empty LI to form expressions of category `sc`, which are selected by *consider*.

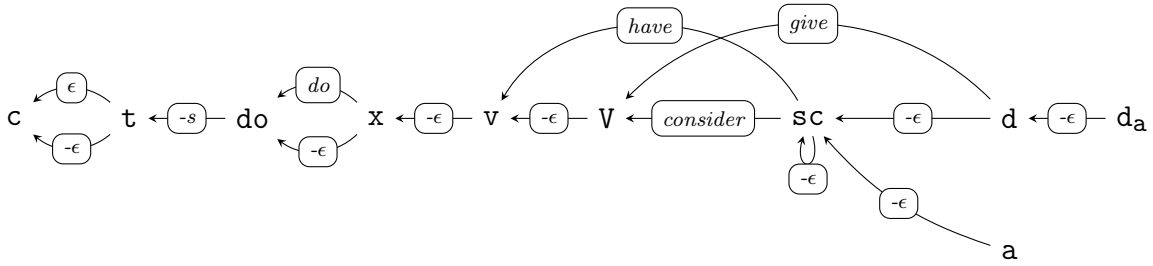
The animacy restriction is implemented by giving animate DPs a category feature distinct from `d`, $\texttt{d}_{\texttt{a}}$. An animate DP can freely become a normal DP by merging with ϵ :: $=\texttt{d}_{\texttt{a}}$ `+k d -k`, but the opposite is not possible. In other words, $\texttt{d}_{\texttt{a}}$ occurs in all contexts that allow `d`, and also in some contexts where `d` is prohibited. The restriction on modification by *why* is added by only allowing *why* to merge with small clauses – expressions of category `sc`. This is done via two LIs: *why* :: `w -wh` and ϵ :: `=sc w= sc`. This fragment allows *why* to modify small clauses but not matrix clauses, since only the former are relevant for the example.[13]

Note that all three grammars are associated with some overgeneration. First, there is no restriction requiring *do*-support in interrogative contexts, which gives rise to examples like **why consider-s John Mary angry*. In addition, all grammars except refined SC (3.14d) treat all small clauses as uniform, producing strings like **John have-s angry* (and, in the case of the uniform small clause analysis, **John give-s Mary angry*). As we have seen before, overgeneration does not affect grammar encoding, but will contribute to a higher cost of encoding some grammatical sentences.

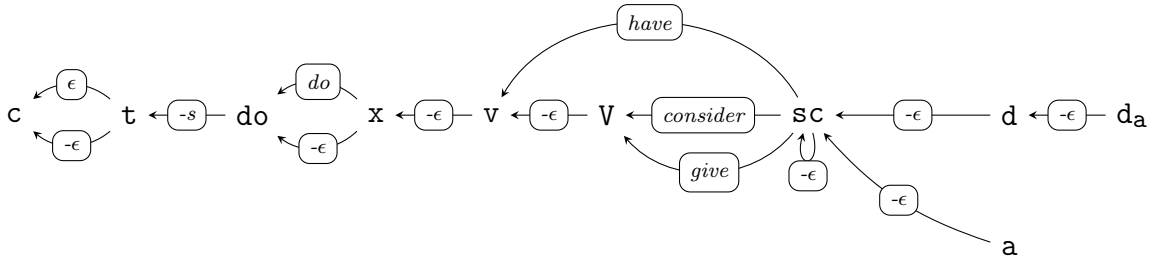
Consider the head-complement graphs in 3.15. The monoclausal *give* (3.15a) selects its arguments directly, whereas the uniform SC *give* (3.15b) takes as its complement the same small clause as *have* and shares its restriction on animacy. On the other hand, the loop at the `sc` vertex represents the position modifiable by *why*. The monoclausal *give* bypasses the category `sc`, unlike *have*; the latter, but not the former, is compatible with *why*. However, the uniform SC *have* merges with expressions of category `sc`, incorrectly allowing modification

13. For the sake of completeness, it would be easy to add modification of matrix clauses by introducing one more empty lexical item: $-\epsilon$ :: `=>v w= v`. Then the grammar would generate different structures corresponding to different readings of *consider*-clauses: *why* [*do-s John consider* [*Mary angry*] ~~*why*~~] vs. *why* [*do-s John consider* [*Mary angry* ~~*why*~~]] (cf. 2a).

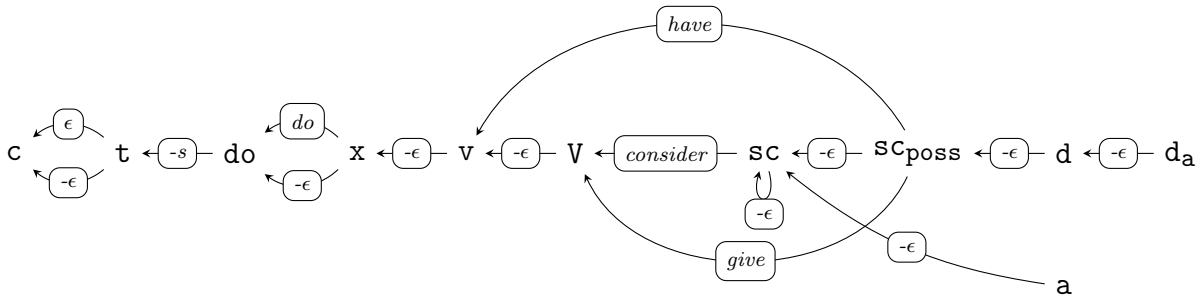
by *why*. Finally, the refined SC analysis (3.15c) gets around both problems by distinguishing between sc and $\texttt{sc}_{\texttt{poss}}$.



(a) Monoclausal



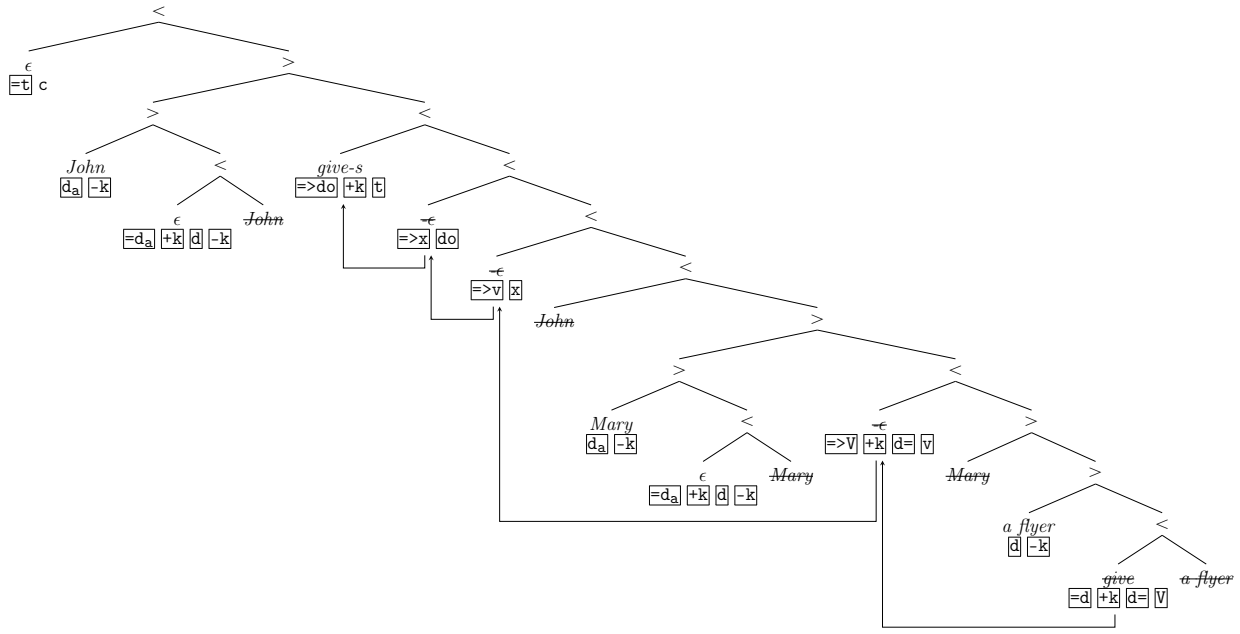
(b) Uniform SC



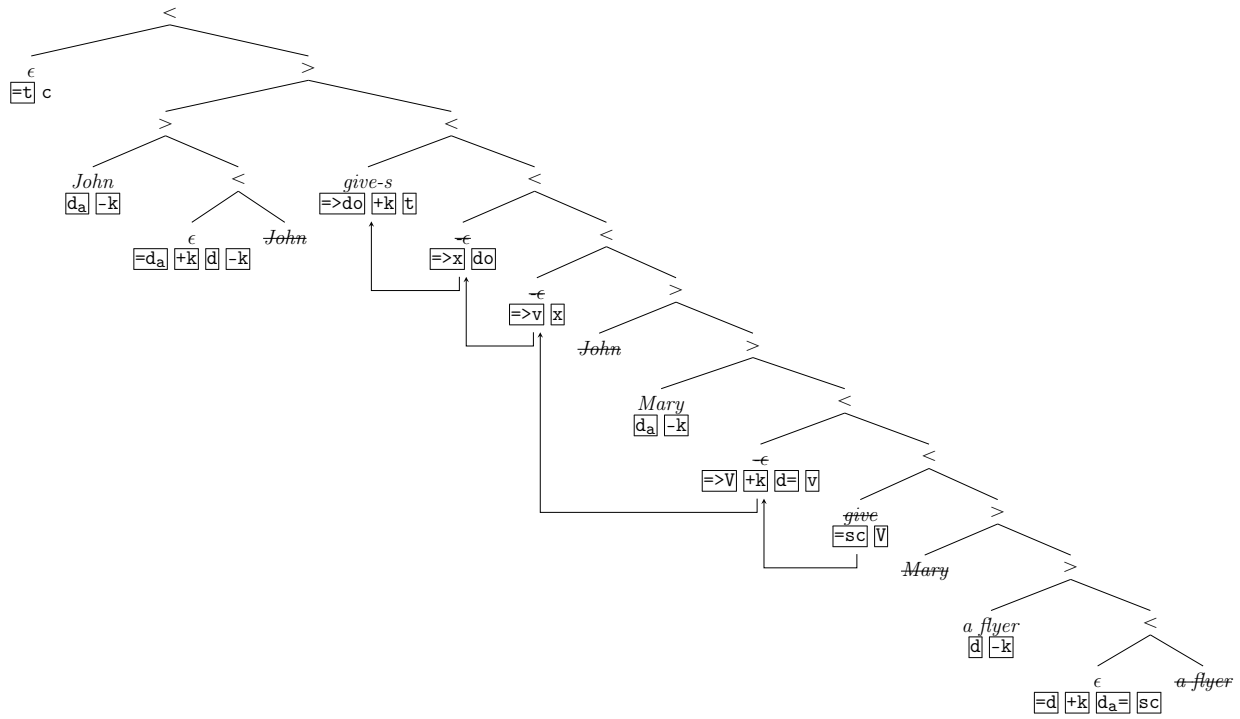
(c) Refined SC

Figure 3.15: Head-complement graphs of MGs in 3.14

As a further illustration, some derived tree examples are given in 3.16.



(a) Monoclausal



(b) Uniform SC

Figure 3.16: Derived trees for *John give-s Mary a flyer*

Grammar encoding costs (Table 3.4) reflect generalizations made by each grammar, as well as the number of category distinctions it makes. Both monoclausal and uniform SC

approaches require 13 distinct categories; however, the latter has a lower cost as it reuses the abstract element heading a small clause, ϵ :: =d +k $\texttt{d}_{\texttt{a}}$= sc, to provide arguments to both *have* and *give*. Refined SCs require an extra category, $\texttt{sc}_{\texttt{poss}}$, as well as an additional lexical item, ϵ :: =$\texttt{sc}_{\texttt{poss}}$ sc, so this grammar ends up having the highest encoding cost.

	$\lvert Base\rvert$	$\sum_{syn}$	$\sum_{phon}$	Grammar (bits)
Monoclausal	13	51	50	955.39
Uniform SC	13	49	50	933.17
Refined SC	14	51	50	966.20

Table 3.4: Grammar metrics for the double-object construction

In order to see how individual analysis choices contribute to corpus encoding, consider the costs of four different sentences shown in Table 3.5. In addition, partial CFGs are given in 3.17. For space reasons, I only include rules with nonzero cost, i.e. those which share the left-hand side with at least one other rule.

	Monoclausal	Uniform SC	Refined SC
John give-s Mary a flyer	$6\log_2 2 + 3\log_2 3$	$7\log_2 2 + 2\log_2 3$	$6\log_2 2 + 2\log_2 3$
	≈ 10.75	≈ 10.17	≈ 9.17
Mary have-s a flyer	$5\log_2 2 + \log_2 3$	$5\log_2 2 + \log_2 3$	$4\log_2 2 + \log_2 3$
	≈ 6.58	≈ 6.58	≈ 5.58
John consider-s Mary angry	$7\log_2 2 + 2\log_2 3$	$7\log_2 2 + 2\log_2 3$	$7\log_2 2 + 2\log_2 3$
	≈ 10.17	≈ 10.17	≈ 10.17
why do-s John consider Mary angry	$6\log_2 2 + 2\log_2 3$	$7\log_2 2 + 2\log_2 3$	$5\log_2 2 + 2\log_2 3$
	≈ 9.17	≈ 10.17	≈ 8.17

Table 3.5: Sentence encoding costs for the double-object construction (bits)

$$
\begin{aligned}
\texttt{(d -k)} &\rightarrow \textit{the car}\\
\texttt{(d -k)} &\rightarrow \textit{a flyer}\\
\texttt{(d -k)} &\rightarrow \texttt{(+k d -k, -k)}\\
\texttt{(d}_\texttt{a}\texttt{ -k)} &\rightarrow \textit{John}\\
\texttt{(d}_\texttt{a}\texttt{ -k)} &\rightarrow \textit{Mary}\\
\texttt{(sc, -k)} &\rightarrow \texttt{(d= sc) (d -k)}\\
\texttt{(v, -k)} &\rightarrow \texttt{(d= v) (d -k)}\\
\texttt{(do, -k)} &\rightarrow \texttt{(=x do) (x, -k)}\\
\texttt{(do, -k)} &\rightarrow \texttt{(=>x do) (x, -k)}\\
\texttt{(c)} &\rightarrow \texttt{(=t c) (t)}\\
\texttt{(c)} &\rightarrow \texttt{(+wh c, -wh)}\\
\texttt{(do, -wh, -k)} &\rightarrow \texttt{(=x do) (x, -wh, -k)}\\
\texttt{(do, -wh, -k)} &\rightarrow \texttt{(=>x do) (x, -wh, -k)}
\end{aligned}
$$

(a) Shared rules

$$
\begin{aligned}
\texttt{(sc, -k)} &\rightarrow \texttt{(d}_\texttt{a}\texttt{= sc) (d}_\texttt{a}\texttt{, -k)}\\
\texttt{(V, -k)} &\rightarrow \texttt{(=sc V) (sc, -k)}\\
\texttt{(V, -k)} &\rightarrow \texttt{(d= V) (d -k)}\\
\texttt{(v, -k)} &\rightarrow \texttt{(=sc v) (sc, -k)}\\
\texttt{(do, -k, -wh)} &\rightarrow \texttt{(=x do) (x, -k, -wh)}\\
\texttt{(do, -k, -wh)} &\rightarrow \texttt{(=>x do) (x, -k, -wh)}\\
\texttt{(t, -wh)} &\rightarrow \texttt{(+k t, -k, -wh)}\\
\texttt{(t, -wh)} &\rightarrow \texttt{(+k t, -wh, -k)}
\end{aligned}
$$

(b) Monoclausal

$$
\begin{aligned}
\texttt{(=sc V)} &\rightarrow \textit{give}\\
\texttt{(=sc V)} &\rightarrow \textit{consider}\\
\texttt{(sc, -k)} &\rightarrow \texttt{(d}_\texttt{a}\texttt{= sc) (d}_\texttt{a}\texttt{, -k)}\\
\texttt{(v, -k)} &\rightarrow \texttt{(=sc v) (sc, -k)}\\
\texttt{(do, -k, -wh)} &\rightarrow \texttt{(=x do) (x, -k, -wh)}\\
\texttt{(do, -k, -wh)} &\rightarrow \texttt{(=>x do) (x, -k, -wh)}\\
\texttt{(t, -wh)} &\rightarrow \texttt{(+k t, -k -wh)}\\
\texttt{(t, -wh)} &\rightarrow \texttt{(+k t, -wh -k)}
\end{aligned}
$$

(c) Uniform SC

$$
\begin{aligned}
\texttt{(sc, -k)} &\rightarrow \texttt{(=>sc}_\texttt{poss}\texttt{ sc) (sc}_\texttt{poss}\texttt{, -k)}\\
\texttt{(V, -k)} &\rightarrow \texttt{(=sc V) (sc, -k)}\\
\texttt{(V, -k)} &\rightarrow \texttt{(=sc}_\texttt{poss}\texttt{ V) (sc}_\texttt{poss}\texttt{, -k)}\\
\texttt{(v, -k)} &\rightarrow \texttt{(=sc}_\texttt{poss}\texttt{ v) (sc}_\texttt{poss}\texttt{, -k)}
\end{aligned}
$$

(d) Refined SC

Figure 3.17: Nonzero-cost CFG rules for 3.14

As expected, the monoclausal approach pays a higher cost to encode examples with *give*, because of its lack of animacy restrictions, whereas the uniform SC grammar overpays for grammatical sentences involving modification by *why*. The third option, refined SCs, does not overpay in either case. In addition, it pays a lower cost to encode *Mary has a flyer*, because of its distinction between small clause types. This corresponds to the fact that this grammar, unlike the other two, does not generate strings like **John has angry*.

For a closer look at individual rules' contribution to these values, let us examine detailed costs of encoding a double object construction, provided in Table 3.6:

	Rule	Cost	Total
Shared rules	(c) → (=t c) (t)	$\log_2 2$	≈ 6.58
	(v, -k) → (d= v) (d -k)	$\log_2 2$	
	(do, -k) → (=x do) (x, -k)	$\log_2 2$	
	(d_a -k) → *John*	$\log_2 2$	
	(d_a -k) → *Mary*	$\log_2 2$	
	(d -k) → *a flyer*	$\log_2 3$	
Monoclausal	(d -k) → (+k d -k, -k)	$2\log_2 3$	≈ 4.17
	(V, -k) → (d= V) (d -k)	$\log_2 2$	
Uniform SC	(d -k) → (+k d -k, -k)	$\log_2 3$	≈ 3.58
	(sc, -k) → (d_a= sc) (d_a, -k)	$\log_2 2$	
	(=sc V) → *give*	$\log_2 2$	
Refined SC	(d -k) → (+k d -k, -k)	$\log_2 3$	≈ 2.58
	(V, -k) → (=sc_{poss} V)	$\log_2 2$	

Table 3.6: Nonzero-cost rules deriving *John give-s Mary a flyer* and their costs (bits)

All three grammars must pay the cost of picking *a flyer* as the object. The monoclausal approach, which lacks animacy restrictions, pays the extra cost of picking an animate Goal, in the form of an additional use of (d -k) → (+k d -k, -k). Next, all three grammars

use a rule to select the right complement type for the verb. However, since the uniform SC grammar assigns the same feature bundle to *give* and *consider*, it has to pay an additional bit to use (=sc V) → *give* and pick the former. Refined SCs pay for each distinction only once, resulting in the lowest cost of encoding the sentence. Thus, the cost of this more complex grammar is offset by the lower cost of encoding the data.

Essentially, what the two positions exemplified by (Harley and Jung 2015) and (Kawakami 2018) disagree on is exactly what properties *have* shares with *give*. MGs can represent these shared properties as syntactic features within LIs which are reused in multiple constructions. This technique offers a way to consider and directly compare insights from multiple literature sources while accounting for possible overgeneration. That said, the grammars examined in this section were constructed by hand. The following chapter takes the next logical step: starting with a naive, theory-neutral grammar and factoring out commonalities shared between lexical items, step by step, to arrive at a better analysis in a principled way.

Chapter 4

Deconstructing syntactic generalizations

4.1 Decomposition of lexical items

Let us re-examine two of the previously discussed minimalist grammars, repeated below:

Mary :: `d -k`	*Mary* :: `d -k`
laughs :: `=d +k t`	*laugh* :: `=d v`
laughed :: `=d +k t`	*jump* :: `=d v`
jumps :: `=d +k t`	*-s* :: `=>v +k t`
jumped :: `=d +k t`	*-ed* :: `=>v +k t`
(a)	(b)

Figure 4.1: Two minimalist grammars (=3.7a, 3.7b)

What does it take to transition from a grammar over words such as 4.1a to a grammar over morphemes such as 4.1b? Given the explicit nature of the formalism, it is relatively easy to keep track of how changes made to a specific LI affect all structures that contain it.

For example, given the lexical item *laughed* :: `=d +k t` (4.2a), we can split the string component into two substrings: *laugh* and *ed*. Similarly, the feature bundle can be split into two subsequences: `=d` and `+k t`. We then assemble two useful lexical items from these elements by introducing `x`, a category feature that is *fresh* (i.e. not used in any lexical item

in the grammar), and combine them into a morphological word with head movement. These lexical items replace the original, as shown in 4.2b. The same operation can be applied even if some or all of the splits result in empty sequences. We can split *laugh* :: `=d x` by assigning its entire string component to one lexical item, and its feature bundle to another (4.2c). This move creates an acategorial root, with only the new feature `y` in its bundle,[14] and shifts its selectional properties to an empty head.

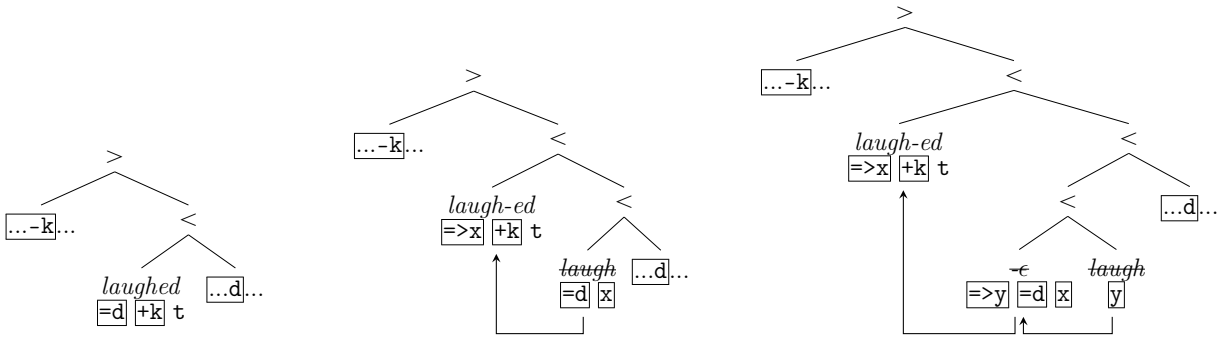


Figure 4.2: Decomposition of *laughed*

Lexical item decomposition is a generalization of this idea proposed by Kobele (2018, to appear). Consider an arbitrary lexical item $w :: \alpha\beta\mathtt{x}\gamma$, where $w \in \Sigma^*$, $\alpha, \beta \in (Sel \cup Lic)^*$, $\mathtt{x} \in Base$, and $\gamma \in Lee^*$. Note that one or both of α, β can be empty. Then decomposition can proceed as shown in 4.3. The original LI is replaced with two new ones, whose string components are u and v, along with a morphological rule generating w from u and v via some morphological operation $\oplus$. The simple model of morphology we adopted in 2.3.2 treats $\oplus$ as string concatenation, eliminating the need to state morphological rules explicitly.

14. The idea that roots are category-neutral and must merge with a category-defining functional head (so every word is at least bimorphemic) is a general assumption in Distributed Morphology adopted, for instance, in (Marantz 1997; Embick and Marantz 2008).

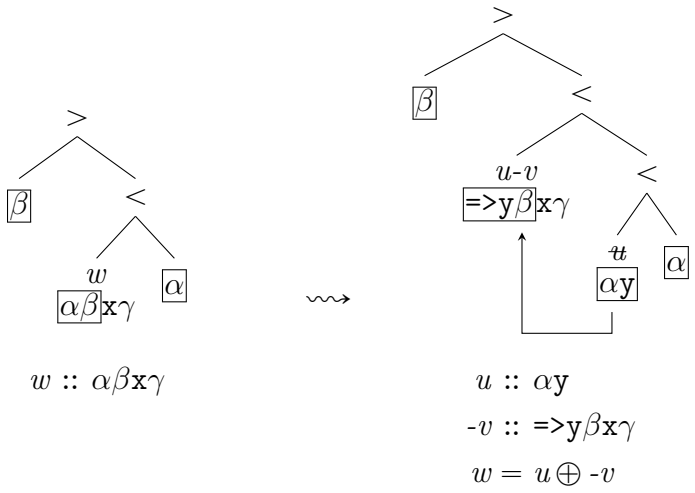


Figure 4.3: Generalized lexical item decomposition

We will refer to the lexical item carrying the fresh category feature as the *lower LI*, and the one selecting it as the *upper LI*.

Recall that every useful lexical item must have a syntactic feature bundle of the following form: $(Sel \cup Lic)^*$ *Base* $(Lee)^*$. Since the resulting structure has to behave as a single unit with respect to **move**, the split must take place to the left of the category feature. An additional restriction arises from our definition of head movement. Since morphological selectors (of type `=>x`) can introduce complements but not specifiers, they are limited to the first feature in the bundle, which means that β may not contain a feature of this type; in other words, the split must occur to the right of the morphological selector, if it is present. Finally, α may not contain any left selectors (`x=`) or overt licensors (`+x`). A decomposition step violating this condition would not produce unusable lexical items, but may cause arguments which originally surfaced to the left of the head to be pronounced to the right instead, as the complex head surfaces in a higher position.[15]

15. This condition preserves linear order by prohibiting any such arguments from being left with the lower LI. A more sophisticated implementation could allow such moves just in case every argument that can possibly be introduced by these features is guaranteed to move further to the left at some point in the derivation.

Lexical decomposition formalizes the process of discovering structure within words. However, it cannot transform a grammar like 3.7a into one like 3.7b on its own. First, decomposition as defined in 4.3 does not generalize. Since every split is associated with a fresh category feature, each newly formed affix is only compatible with a single stem. To remedy this, we need some way of unifying existing categories. Second, decomposing items in isolation is extremely permissive. String components can be split anywhere, and feature bundles anywhere between the `=>x`-type selector (if present) and the category. For example, *laughed* :: `=d +k t` can be decomposed in 24 ways (assuming concatenative morphology), of which only a few are linguistically reasonable. Whether a given decomposition step is worthwhile depends on other items in the lexicon.

In order to recognize useful steps and make generalizations, decomposition needs to process multiple lexical items simultaneously, using similarities between them to inform its decisions. We will refer to this strategy as *batch decomposition*. Consider the transition in 4.4. It starts with a batch of four lexical items and factors out the elements they have in common: the prefix *laugh* and the syntactic feature `=d`. These repeated elements are then expressed as a new lexical item, *laugh* :: `=d x`, which is reused for every word in the batch, adding only one new feature to *Base*. This is an example of *left decomposition*, as the shared part of both the string component and feature bundle is on the left-hand side.

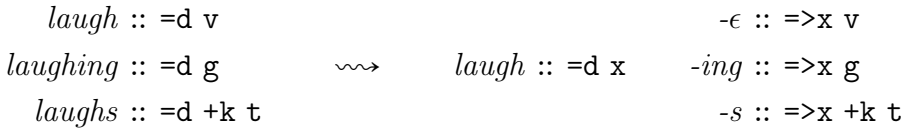


Figure 4.4: Left decomposition

We can also obtain meaningful generalizations via *right decomposition*, by factoring out right-hand side commonalities (4.5). In a (primarily) suffixal language, left decomposition can be used to identify stems, and right decomposition to identify affixes.

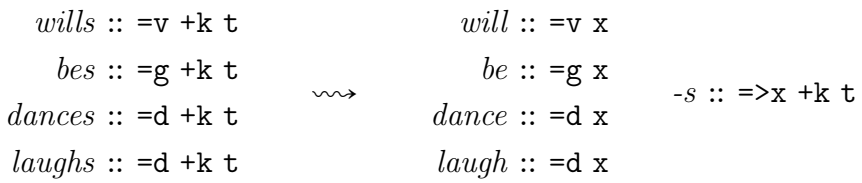


Figure 4.5: Right decomposition

Note that all previously defined restrictions on decomposition must still apply to every lexical item in a batch. In particular, left decomposition does not necessarily require the batch to have anything in common, as both u and α can be empty. Right decomposition, on the other hand, is only applicable if the batch shares at least the category and postcategorial features.

Definition 4.1: Left decomposition

Let $l_{phon}, l_{syn} \in \mathbb{N}$. Let $Batch = \{uv_1 :: \alpha\beta_1, ..., uv_n :: \alpha\beta_n\} \subseteq Lex$, where $u, v_1...v_n \in \Sigma^*$; $\alpha \in (Sel \cup Lic)^*$, $\beta_1...\beta_n \in Syn^*$, such that $|u| = l_{phon}$ and $|\alpha| = l_{syn}$. Then *left decomposition* of $Batch$ in Lex at l_{phon} and l_{syn} produces the following lexicon:

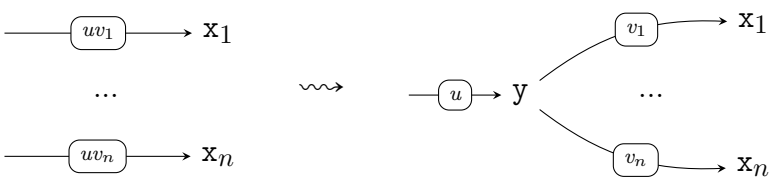
$\mathbf{ldec}(Lex, Batch, l_{phon}, l_{syn}) = (Lex - Batch) \cup \{u :: \alpha\mathtt{y}, v_1 :: \mathtt{=>y}\beta_1, ..., v_n :: \mathtt{=>y}\beta_n\}$, where $\mathtt{y} \notin Base$.

Definition 4.2: Right decomposition

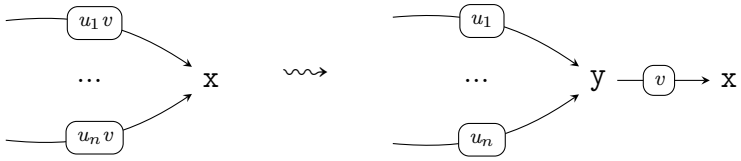
Let $l_{phon}, l_{syn} \in \mathbb{N}$. Let $Batch = \{u_1v :: \alpha_1\beta, ..., u_nv :: \alpha_n\beta\} \subseteq Lex$, where $u_1...u_n, v \in \Sigma^*$; $\alpha_1...\alpha_n \in (Sel \cup Lic)^*$; $\beta \in Syn^*$, such that $|v| = l_{phon}$ and $|\beta| = l_{syn}$. Then *right decomposition* of $Batch$ in Lex at l_{phon} and l_{syn} produces the following lexicon:

$\mathbf{rdec}(Lex, Batch, l_{phon}, l_{syn}) = (Lex - Batch) \cup \{u_1 :: \alpha_1\mathtt{y}, ..., u_n :: \alpha_n\mathtt{y}, v :: \mathtt{=>y}\beta\}$, where $\mathtt{y} \notin Base$.

Since both left and right batch decomposition introduce new head-complement relations by creating complex words, their effect can be easily seen on graphs, as shown in 4.6.



(a) Left



(b) Right

Figure 4.6: Generalized batch decomposition

Left decomposition (4.6a) replaces a set of edges (which may or may not share the start and/or end vertex) with paths which all pass through the same vertex corresponding to the new feature. Right decomposition (4.6b) is similar, except the original edges are required to share the end vertex.

4.2 Auxiliary operations

Consider a certain type of silent LIs that can be formed by decomposition – namely, those of the form $\text{-}\epsilon$ :: =>x y, for any $\mathtt{x}, \mathtt{y} \in \mathit{Base}$. We will refer to them as *category changers*. As mentioned in Subsection 2.3.1, such LIs encode the idea that some syntactic categories have a more limited distribution than others. Having $\text{-}\epsilon$ :: =>x y in the lexicon means that any expression of category x can freely become one of category y without additional changes to its phonetic or syntactic content, whereas the opposite is not necessarily possible. A common use of category changers is to implement a hierarchy of projections in a minimalist grammar without additional machinery.

That said, in certain cases category changers are redundant in a way that cannot be eliminated by further decomposition steps. This happens if one of the following is true:

- there is no reason to distinguish between `x` and `y`;
- the relationship between `x` and `y` is already expressed elsewhere in the lexicon.

To examine the first type of redundancy, consider the minimalist grammar in 4.7 which includes, among other LIs, the decomposed batch shown in 4.4. In this case, the distinction between `x` and `v` is an unnecessary one. The two categories can be unified, and the category changer removed from the lexicon, without any changes to the weak generative capacity of the grammar.

Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
-ϵ :: `=>x v`
-ing :: `=>x g`
-s :: `=>x +k t`
laugh :: `=d x`

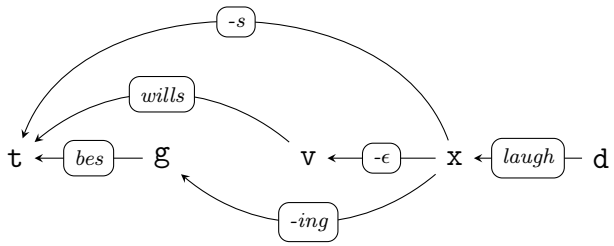


Figure 4.7: MG containing the batch from 4.4

In general, we can eliminate a category changer -ϵ :: `=>x y` by removing it from the grammar and replacing all remaining instances of `x` and `y` with a fresh category `z`. This essentially reverses the effects of decomposition for a specific lexical item in the batch. In graph terms, this is *edge contraction*, an operation which removes an edge and merges the two vertices it previously joined.

Definition 4.3: Feature renaming

Let $\mathtt{x}_1, ..., \mathtt{x}_k, \mathtt{y}_1, ..., \mathtt{y}_k \in \mathit{Base}$, $\delta \in \mathit{Syn}^*$. Then $\delta[\mathtt{x}_1 \mapsto \mathtt{y}_1, ..., \mathtt{x}_k \mapsto \mathtt{y}_k]$ is the feature bundle which is identical to δ except for each $\mathtt{x}_i$ replaced with $\mathtt{y}_i$, for $i \in [1, k]$.

Extending this definition to grammars, let $\mathit{Lex} = \{s_1 :: \delta_1, ..., s_n :: \delta_n\}$, where $s_1, ..., s_n \in \Sigma^*$, $\delta_1, ..., \delta_n \in \mathit{Syn}^*$. Then $\mathit{Lex}[\mathtt{x}_1 \mapsto \mathtt{y}_1, ..., \mathtt{x}_k \mapsto \mathtt{y}_k] = \{s_1 :: \delta_1[\mathtt{x}_1 \mapsto \mathtt{y}_1, ..., \mathtt{x}_k \mapsto \mathtt{y}_k], ..., s_n :: \delta_n[\mathtt{x}_1 \mapsto \mathtt{y}_1, ..., \mathtt{x}_k \mapsto \mathtt{y}_k]\}$.

Definition 4.4: Edge contraction

Let $c =$ `-ϵ :: =>x y` $\in \mathit{Lex}$, where $\mathtt{x}, \mathtt{y} \in \mathit{Base}$. Then *edge contraction* of c in Lex produces the following lexicon:

$\mathbf{con}(\mathit{Lex}, c) = (\mathit{Lex} - \{c\})[\mathtt{x} \mapsto \mathtt{z}, \mathtt{y} \mapsto \mathtt{z}]$, where $\mathtt{z} \notin \mathit{Base}$.

Unlike decomposition, edge contraction comes with an important caveat. Without restrictions, this operation can create new paths in the grammar, potentially causing it to overgenerate. An obvious example is the case when there exists another path from `x` to `y`. If both are replaced with `z`, it becomes a cycle. Figure 4.8 offers a more general description of the problem:

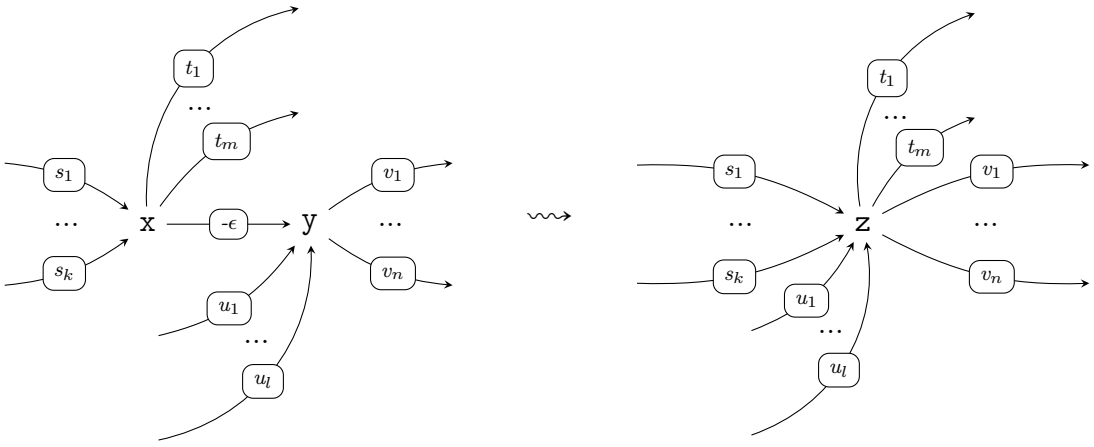


Figure 4.8: Generalized edge contraction

Let $s_1, ..., s_k$ and $u_1, ..., u_l$ represent the string components of lexical items corresponding to edges whose destination is, respectively, `x` and `y`. Similarly, let $t_1, ..., t_m$ and $v_1, ..., v_n$ stand for edges with origin in `x` and `y`. Any of the $s_1, ..., s_k$ may be selected (directly or via the category changer -ϵ :: =>`x y`) by any of the $v_1, ..., v_n$; however, only $s_1, ..., s_k$ may be selected by $t_1, ..., t_m$. Once the category changer is contracted and both `x` and `y` renamed to `z`, this asymmetry is eliminated. All of $u_1, ..., u_l$ become compatible with $t_1, ..., t_m$, producing new paths and potentially new words.

The second source of redundancy is found in grammars where the same relationship between categories is expressed by multiple lexical items. Consider the following abstract example:

-ϵ :: =>`x y`
-ϵ :: =>`x z`
-ϵ :: =>`z y`

Regardless of any other LIs that may be present in the lexicon, any expression selected by -ϵ :: =>`x y` can instead be selected by -ϵ :: =>`x z`, forming, in turn, a valid complement for -ϵ :: =>`z y`. Because of this, -ϵ :: =>`x y` is unnecessary and can be safely removed from the lexicon. This can be extended to cases where the alternative to a given LI contains any number of other category changers, including the special case of a pair of completely identical LIs, one of which may be deleted. We formalize this as another auxiliary operation, *edge deletion*, which targets category changers whose usage can be emulated with one or more other category changers also present in the lexicon. Unlike edge contraction, edge deletion does not involve feature conflation or renaming and poses no risk of overgeneration.

Definition 4.5: Edge deletion

Let $c =$ -ϵ :: =>$\mathtt{x}_1$ $\mathtt{x}_n \in$ *Lex*, where $\mathtt{x}_1, ..., \mathtt{x}_n \in$ *Base* and for $i \in [1, n-1]$, -ϵ :: =>$\mathtt{x}_i$ $\mathtt{x}_{i+1}$ distinct from c is in *Lex*. Then *edge deletion* of c in *Lex* produces the following lexicon: **del**$(Lex, c) = Lex - \{c\}$.

From the graph perspective, a category changer -ϵ :: =>$\mathtt{x}_1$ $\mathtt{x}_n$ is a candidate for edge deletion if there is another path from $\mathtt{x}_1$ to $\mathtt{x}_n$ where all edges are category changers as well. This condition ensures that any word using -ϵ :: =>$\mathtt{x}_1$ $\mathtt{x}_n$ can also be generated without it.

Figure 4.9: Generalized edge deletion

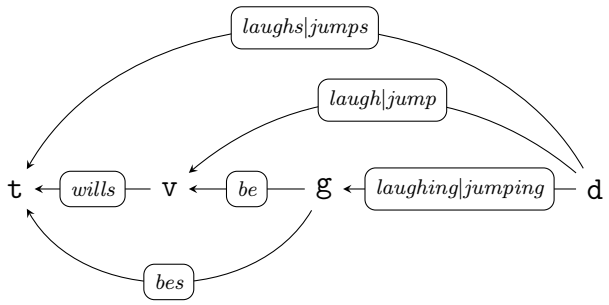
4.3 Transforming a grammar

The following example shows how the three operations introduced in the previous section can transform a naive word-based grammar into a linguistically motivated grammar over morphemes. We start with a corpus of eight sentences below. As before, we abstract away from morphological irregularities of English and replace each word with a sequence of morphemes formed by string concatenation; *is* is rendered as *bes*, and *will* as *wills*.

Mary laughs;
Mary jumps;
Mary bes laughing;
Mary bes jumping;
Mary wills laugh;
Mary wills jump;
Mary wills be laughing;
Mary wills be jumping.

These sentences are generated by the small lexicon of unsegmented words encoding a fragment of the English auxiliary system shown in 4.10. For the sake of space, LIs with identical feature bundles in the graph representation are depicted as a single edge labeled with all relevant string components separated with |. We will continue using metrics introduced in Chapter 3 to keep track of changes introduced by each transition.

Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
laughs :: `=d +k t`
laughing :: `=d g`
laugh :: `=d v`
jumps :: `=d +k t`
jumping :: `=d g`
jump :: `=d v`



$\lvert Base \rvert$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
5	24	49	565.54	24.68

Figure 4.10: Original lexicon

This grammar's full CFG counterpart is given in 4.11, with some sample derivations provided in 4.12.

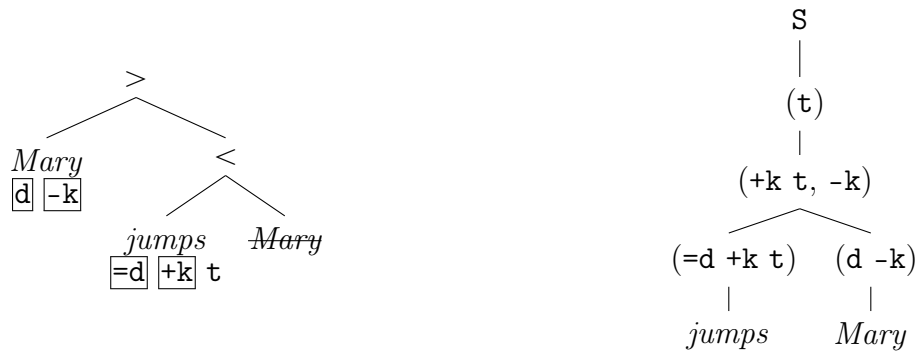
$$
\begin{aligned}
\texttt{S} &\rightarrow (\texttt{t}) \\
(\texttt{t}) &\rightarrow (\texttt{+k t, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=d +k t})\ (\texttt{d -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=g +k t})\ (\texttt{g, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=v +k t})\ (\texttt{v, -k}) \\
(\texttt{g, -k}) &\rightarrow (\texttt{=d g})\ (\texttt{d -k}) \\
(\texttt{v, -k}) &\rightarrow (\texttt{=d v})\ (\texttt{d -k}) \\
(\texttt{v, -k}) &\rightarrow (\texttt{=g v})\ (\texttt{g, -k})
\end{aligned}
$$

$$
\begin{aligned}
(\texttt{d -k}) &\rightarrow \textit{Mary} \\
(\texttt{=g +k t}) &\rightarrow \textit{bes} \\
(\texttt{=v +k t}) &\rightarrow \textit{wills} \\
(\texttt{=g v}) &\rightarrow \textit{be} \\
(\texttt{=d +k t}) &\rightarrow \textit{laughs} \\
(\texttt{=d g}) &\rightarrow \textit{laughing} \\
(\texttt{=d v}) &\rightarrow \textit{laugh} \\
(\texttt{=d +k t}) &\rightarrow \textit{jumps} \\
(\texttt{=d g}) &\rightarrow \textit{jumping} \\
(\texttt{=d v}) &\rightarrow \textit{jump}
\end{aligned}
$$

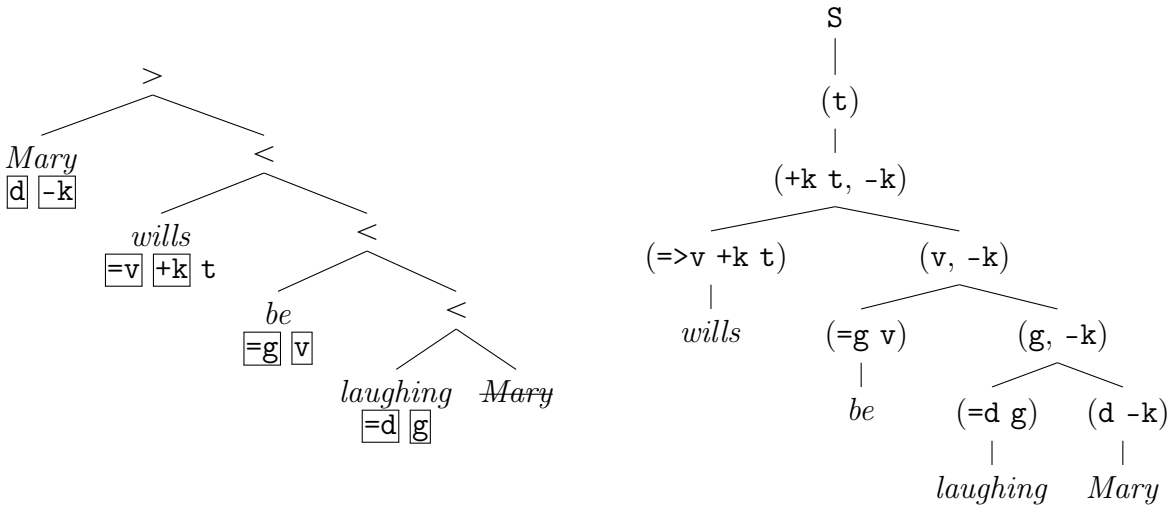
Figure 4.11: CFG counterpart of 4.10

In this particular case, all choices made in the course of the derivation are limited to complement selection. Because of this, the cost of each sentence can also be read off the head-complement graph by following the path from `t` to `d` (i.e. in the top-down direction) and assigning an appropriate cost in bits to each node that has more than one incoming arc. For example, it is easy to see that each sentence has to pay $\log_2 3 \approx 1.58$ bits to

rewrite (+k t, -k) – which corresponds to the three arcs entering the vertex t on the graph. In addition, each sentence pays one bit to select between the two lexical verbs (*laugh* vs. *jump*). However, any sentence that uses *wills* also pays an extra bit to rewrite (v, -k). Because this grammar operates over unsegmented words, the auxiliary *be* is implemented as two distinct lexical items, *bes* :: =g +k t and *be* :: =g v. The cost of having the progressive construction as an option is paid twice: first by each sentence in the corpus as *bes* :: =g +k t necessitates the CFG rule (+k t, -k) → (=g +k t) (g, -k), and again by sentences involving *wills* as *be* :: =g v necessitates the CFG rule (v, -k) → (=g v) (g, -k). This straightforward implementation is therefore suboptimal with respect to the corpus cost.



(a) *Mary jumps*



(b) *Mary wills be laughing*

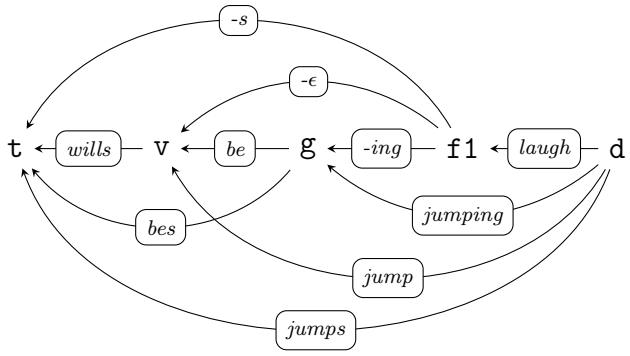
Figure 4.12: Sample derived trees and CFG parse trees (4.10, 4.11)

We begin by decomposing lexical verbs. The following lexical items are a good target for left decomposition, as they share the prefix *laugh* and the first syntactic feature `=d`:

laughs :: `=d +k t`
laughing :: `=d g`
laugh :: `=d v`

By factoring out these phonological and syntactic commonalities and associating them with the fresh feature `f1`, we arrive at the lexicon in 4.13.

Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
-s :: `=>f1 +k t`
-ing :: `=>f1 g`
-ϵ :: `=>f1 v`
laugh :: `=d f1`
jumps :: `=d +k t`
jumping :: `=d g`
jump :: `=d v`



$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
6	26	39	542.84	26.34

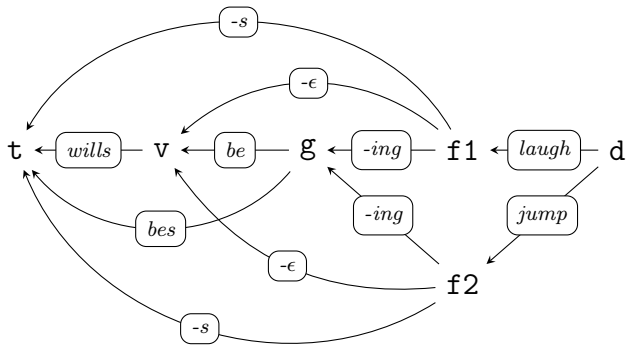
Figure 4.13: Decomposition of *laugh*

In a similar way, we target the other verbal paradigm in the lexicon and factor out the prefix *jump* and the feature `=d` via left decomposition:

jumps :: `=d +k t`
jumping :: `=d g`
jump :: `=d v`

This move introduces another fresh feature, `f2`, and results in two copies each of the affixes *-s*, *-ϵ*, and *-ing* (4.14).

Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
-s :: `=>f1 +k t`
-ing :: `=>f1 g`
-ϵ :: `=>f1 v`
laugh :: `=d f1`
-s :: `=>f2 +k t`
-ing :: `=>f2 g`
-ϵ :: `=>f2 v`
jump :: `=d f2`



$\lvert Base \rvert$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
7	28	31	530.40	26.34

Figure 4.14: Decomposition of *jump*

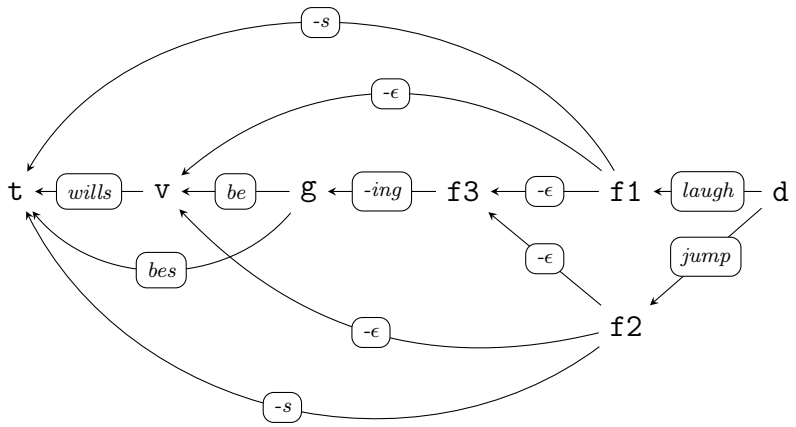
Note that the corpus cost has increased. There are now four arcs entering `t`, which translate to two bits paid by each sentence. Sentences that involve the category `v` still pay the $\log_2 3$ bits, and those involving `g` pay an extra bit to pick one of the two paths labeled with *-ing*.

Next we unify *-ing* :: `=>f1 g` and *-ing* :: `=>f2 g` by making a right decomposition step. Since these lexical items share the entire string component, we can associate their syntactic differences, `=>f1` vs. `=>f2`, with empty heads (4.15). Each of these transitions immediately pays off in terms of $\sum_{phon}$, but adds a new category feature to *Base* and increases $\sum_{syn}$.

At this point, both *-ϵ* :: `=>f1 f3` and *-ϵ* :: `=>f2 f3` are valid targets for edge contraction. This move collapses remaining distinctions between *laugh* and *jump*, assuming that they have identical syntactic distribution. The new projection introduced by `f4` now hosts both lexical verbs (4.16). Even though *jump* can now be selected by affixes previously compatible only with *laugh*, and vice versa, no new morphological words are created.[16]

16. Not every possible contraction step has this property. For example, contracting *-ϵ* :: `=>f1 v` instead would create a path from `v` back to `f3`, produce a cycle in morphotactics, and generate the infinite set of ungrammatical sentences **Mary will-s be* (*be-ing*)$^+$ *laugh-ing*.

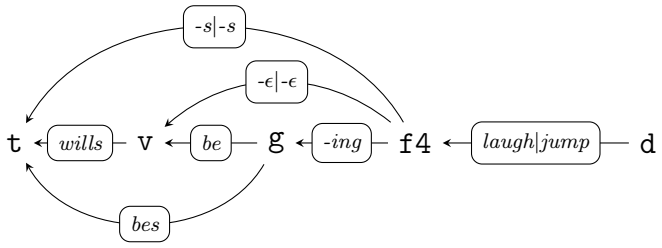
Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
-s :: `=>f1 +k t`
-ing :: `=>f3 g`
-ϵ :: `=>f1 f3`
-ϵ :: `=>f1 v`
laugh :: `=d f1`
-s :: `=>f2 +k t`
-ϵ :: `=>f2 v`
-ϵ :: `=>f2 f3`
jump :: `=d f2`



$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
8	30	28	544.62	26.34

Figure 4.15: Decomposition of *-ing*

Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
-s :: `=>f4 +k t`
-ing :: `=>f4 g`
-ϵ :: `=>f4 v`
laugh :: `=d f4`
-s :: `=>f4 +k t`
-ϵ :: `=>f4 v`
jump :: `=d f4`



$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
6	26	28	484.30	28.68

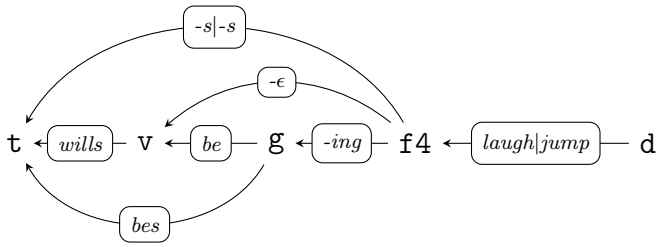
Figure 4.16: Contraction of `f1` → `f3` and `f2` → `f3`

This move reduces the grammar cost; however, the corpus cost sees an increase. This is due to two pairs of parallel edges, or duplicate lexical items. The distinctions between the lexical verbs have been collapsed, but the grammar still contains a separate *-ϵ* :: `=>f4 v` and

-*s* :: `=>f4 +k t` contributed by each of them. In short, any sentence involving these LIs has to pay the 1-bit cost of picking the lexical verb twice.

We can remove one copy of -ϵ :: `=>f4 v` as an instance of edge deletion (4.17).

Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
-*s* :: `=>f4 +k t`
-*ing* :: `=>f4 g`
laugh :: `=d f4`
-ϵ :: `=>f4 v`
-*s* :: `=>f4 +k t`
jump :: `=d f4`



$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
6	24	28	457.69	26.68

Figure 4.17: Deletion of a duplicate edge `f4` → `v`

This reduces both $\sum_{syn}$ and the corpus cost. Our definition of this operation only extends to category changers, which prevents it from targeting -*s* :: `=>f4 +k t` at this time.

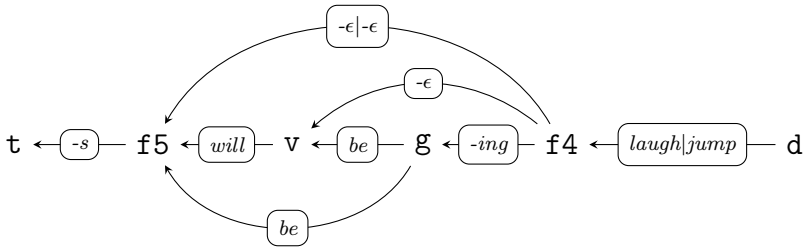
We turn next to the following four items, two of which are identical:

bes :: `=g +k t`
wills :: `=v +k t`
-*s* :: `=>f4 +k t`
-*s* :: `=>f4 +k t`

This batch shares the common suffix *s* and the sequence of syntactic features `+k t`. By factoring out these similarities, we create what is essentially a dedicated Tense projection (4.18). Since the batch shares more than one syntactic feature, this decomposition step immediately reduces $\sum_{syn}$ as well as $\sum_{phon}$.

This move reduces the two duplicate items we have had since 4.16 to category changers. We are now in position to get rid of one of the copies by deleting one of the parallel edges from `f4` to `f5` (4.19).

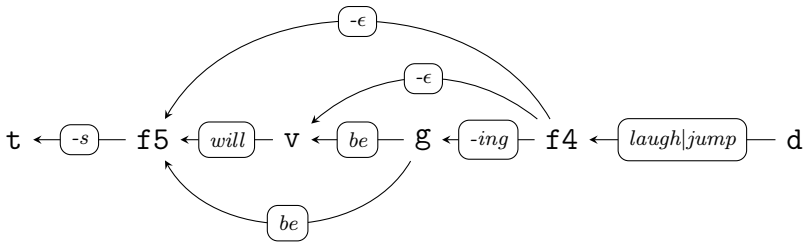
Mary :: `d -k`
-s :: `=>f5 +k t`
be :: `=g f5`
will :: `=v f5`
be :: `=g v`
-ϵ :: `=>f4 f5`
-ing :: `=>f4 g`
-ϵ :: `=>f4 v`
laugh :: `=d f4`
-ϵ :: `=>f4 f5`
jump :: `=d f4`



$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
7	23	25	439.32	26.68

Figure 4.18: Decomposition of Tense

Mary :: `d -k`
-s :: `=>f5 +k t`
be :: `=g f5`
will :: `=v f5`
be :: `=g v`
-ϵ :: `=>f4 f5`
-ing :: `=>f4 g`
-ϵ :: `=>f4 v`
laugh :: `=d f4`
jump :: `=d f4`



$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
7	21	25	412.53	24.68

Figure 4.19: Deletion of a duplicate edge `f4` → `f5`

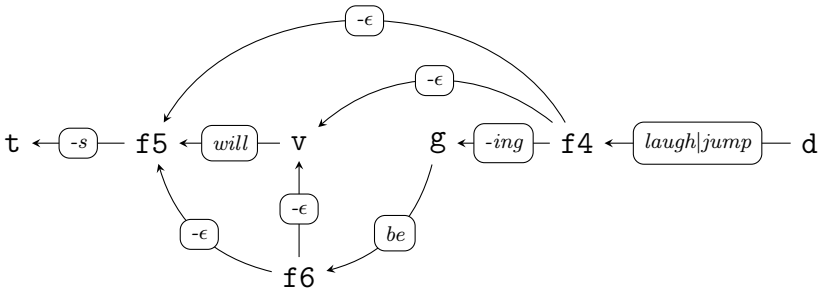
Once again, both $\sum_{syn}$ and the corpus cost go down without increases to other metrics. The corpus cost drops back to that of the original, 24.68 bits.

The grammar still contains two partially redundant lexical items:

be :: `=g f5`
be :: `=g v`

As before, we apply batch decomposition to deal with them (4.20).

Mary :: `d -k`
-s :: `=>f5 +k t`
-ϵ :: `=>f6 f5`
be :: `=g f6`
will :: `=v f5`
-ϵ :: `=>f6 v`
-ϵ :: `=>f4 f5`
-ing :: `=>f4 g`
-ϵ :: `=>f4 v`
laugh :: `=d f4`
jump :: `=d f4`

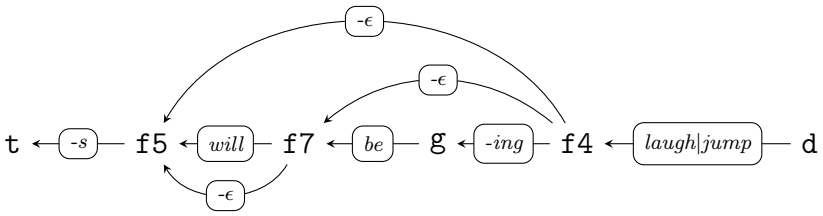


$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
8	23	23	431.39	24.68

Figure 4.20: Decomposition of *be*

We then contract the category changer *-ϵ* :: `=>f6 v` (4.21).[17] This move brings $|Base|$ back to 7 and $\sum_{syn}$ back to 21 – same as before we started decomposing the paradigm of *be* (4.19), while retaining the reduced $\sum_{phon}$.

Mary :: `d -k`
-s :: `=>f5 +k t`
-ϵ :: `=>f7 f5`
be :: `=g f7`
will :: `=f7 f5`
-ϵ :: `=>f4 f5`
-ing :: `=>f4 g`
-ϵ :: `=>f4 f7`
laugh :: `=d f4`
jump :: `=d f4`



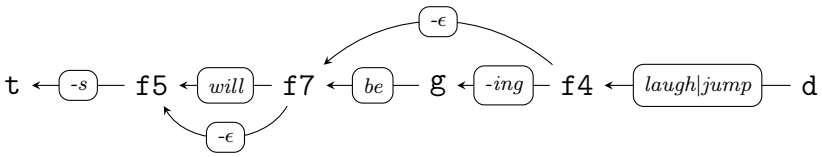
$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
7	21	23	401.82	26.68

Figure 4.21: Contraction of `f6` → `v`

17. Note that we should not do the same to *-ϵ* :: `=>f6 f5`, since that would result in an undesirable cycle generating **Mary will-s* (*will*)$^+$ *be laugh-ing*.

Collapsing `f6` and `v` into `f7` has created a path of category changers from `f4` to `f5`. Having two paths leading to the same output string increases the corpus cost yet again. At the same time, -ϵ :: `=>f4 f5` now meets the condition for edge deletion (4.22).

Mary :: `d -k`
-*s* :: `=>f5 +k t`
-ϵ :: `=>f7 f5`
be :: `=g f7`
will :: `=f7 f5`
-*ing* :: `=>f4 g`
-ϵ :: `=>f4 f7`
laugh :: `=d f4`
jump :: `=d f4`



$\|Base\|$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus
7	19	23	375.03	24.00

Figure 4.22: Deletion of `f4` → `f5`

This last step does not change $|Base|$ or $\sum_{phon}$, but finally pushes $\sum_{syn}$ below the value it had in 4.19. In addition, the total corpus cost is reduced to 24 bits. Thus, the decision made in 4.20 to decompose the two instances of *be* rather than leave them intact took two steps to fully pay off.

The final lexicon (4.22) captures a number of correct generalizations, to the extent they can be encoded with a minimalist grammar. In particular, the roots in both verbal paradigms have been identified and separated from inflectional morphology, and they share the same syntactic category. The grammar also contains three category changers, silent lexical items which impose a hierarchy on category features – for instance, a lexical verb can be selected (directly or via category changers) by -*ing*, the modal *will*, or the Tense marker -*s*, whereas a phrase headed by *will* can only be selected by -*s*. These changes to the lexicon are reflected in the metrics, as shown in Table 4.1.

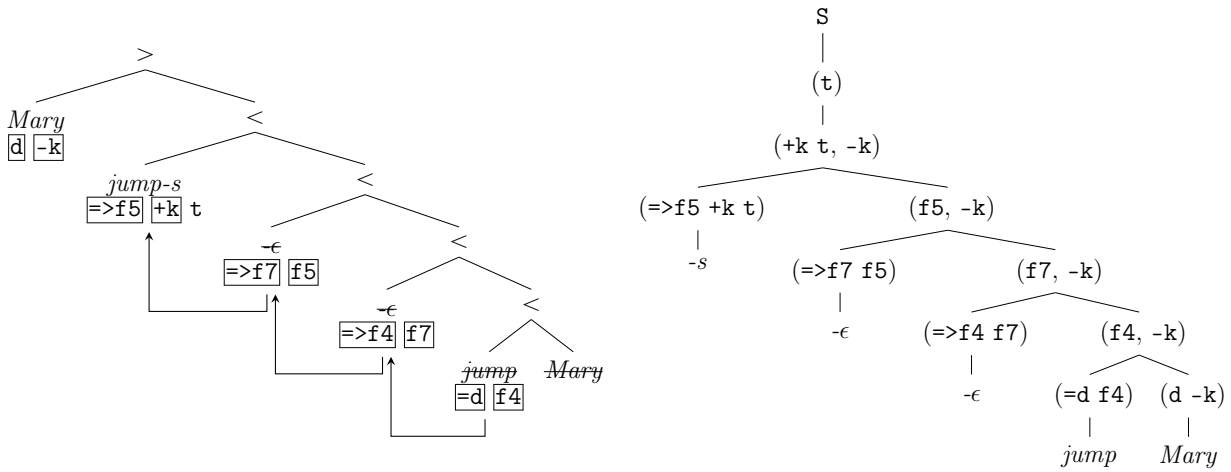
	$\lvert Base \rvert$	$\sum_{syn}$	$\sum_{phon}$	Grammar	Corpus	MDL
Original (4.10)	5	24	49	565.54	24.68	590.22
Final (4.22)	7	19	23	375.03	24.00	399.03

Table 4.1: Original lexicon vs. final lexicon

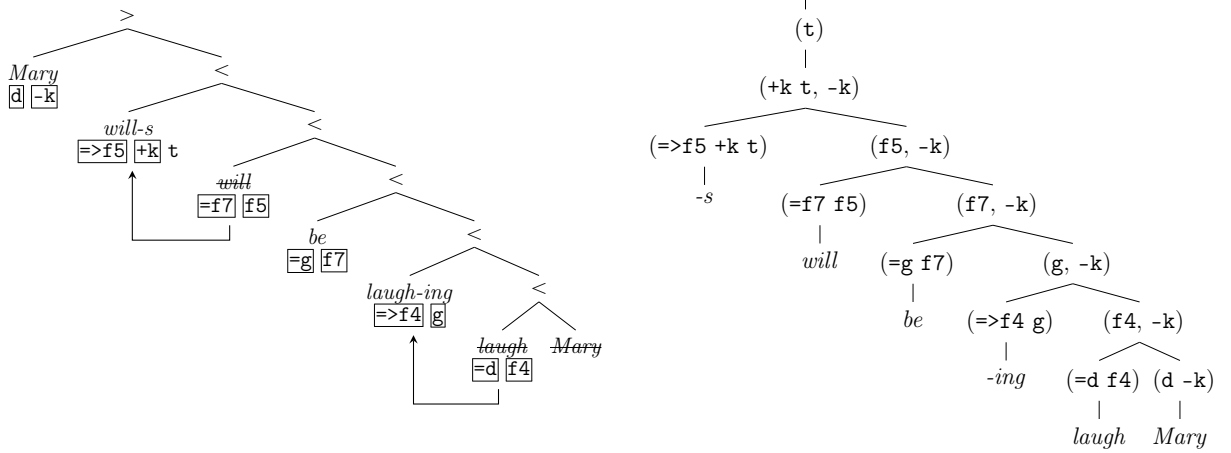
$$
\begin{aligned}
\texttt{S} &\rightarrow (\texttt{t}) \\
(\texttt{t}) &\rightarrow (\texttt{+k t, -k}) \\
(\texttt{+k t, -k}) &\rightarrow (\texttt{=>f5 +k t}) \ (\texttt{f5, -k}) \\
(\texttt{f5, -k}) &\rightarrow (\texttt{=>f7 f5}) \ (\texttt{f7, -k}) \\
(\texttt{f5, -k}) &\rightarrow (\texttt{=f7 f5}) \ (\texttt{f7, -k}) \\
(\texttt{f7, -k}) &\rightarrow (\texttt{=>f4 f7}) \ (\texttt{f4, -k}) \\
(\texttt{f7, -k}) &\rightarrow (\texttt{=g f7}) \ (\texttt{g, -k}) \\
(\texttt{g, -k}) &\rightarrow (\texttt{=>f4 g}) \ (\texttt{f4, -k}) \\
(\texttt{f4, -k}) &\rightarrow (\texttt{=d f4}) \ (\texttt{d -k})
\end{aligned}
\qquad
\begin{aligned}
(\texttt{d -k}) &\rightarrow \textit{Mary} \\
(\texttt{=>f5 +k t}) &\rightarrow \textit{-s} \\
(\texttt{=>f7 f5}) &\rightarrow \textit{-}\epsilon \\
(\texttt{=g f7}) &\rightarrow \textit{be} \\
(\texttt{=f7 f5}) &\rightarrow \textit{will} \\
(\texttt{=>f4 g}) &\rightarrow \textit{-ing} \\
(\texttt{=>f4 f7}) &\rightarrow \textit{-}\epsilon \\
(\texttt{=d f4}) &\rightarrow \textit{laugh} \\
(\texttt{=d f4}) &\rightarrow \textit{jump}
\end{aligned}
$$

Figure 4.23: CFG counterpart of 4.22

The final grammar is an improvement over the original in terms of both $\sum_{syn}$ and $\sum_{phon}$, at the cost of two new features added to *Base*. Most of the derivations take more steps than their counterpart in the original grammar (4.24). However, the cost of encoding each sentence is now exactly three bits (cf. 4.23), which brings the total corpus cost down to 24 bits.



(a) *Mary jump-s*



(b) *Mary will-s be laugh-ing*

Figure 4.24: Sample derived trees and CFG parse trees (4.22, 4.23)

4.4 Interim summary

As mentioned earlier, the task undertaken in this dissertation can be characterized as a search problem, which has two main components:

- identify the search space of possible solutions;
- examine the search space to find the best solution.

Focusing primarily on the former, this chapter introduced a set of operations over grammars centered around the idea of batch decomposition, which formalizes the intuition behind linguistic generalizations. If decomposing a single LI captures discovering morphological structure within a word, batch decomposition identifies redundancies within the lexicon and factors them out as shared morphemes. With this in mind, denoting the starting lexicon as Lex_{start}, we can define the search space as the closure of $\{Lex_{start}\}$ under batch decomposition, edge contraction, and edge deletion. This set is infinite thanks to decomposition; for example, an arbitrary lexical item s :: α `x` γ can be trivially decomposed into s :: α `y` and -ϵ :: `=>y x` γ.

The main question associated with the first component of the search problem is whether the solution we are looking for is, in fact, in the search space – in this case, whether there exists a sequence of batch decomposition, edge contraction, and edge deletion steps leading to the desired result. As demonstrated by 4.10–4.22, these operations do allow one to transform a grammar over words into a linguistically motivated grammar over morphemes through a series of well-defined steps. In this example, the final grammar (4.22) was produced by using left decomposition to identify members of the same paradigm and right decomposition to make generalizations across paradigms, while letting edge contraction and deletion take care of any redundancies arising in the process.

Turning to the second component, in Chapter 2 we discussed a way of measuring the complexity of a minimalist grammar with respect to a given dataset: the three basic metrics ($Base$, $\sum_{syn}$, and $\sum_{phon}$), the cost of encoding the corpus represented as a set of CFG derivations, and the MDL measure which takes all of these into account. This approach makes explicit a trade-off: a grammar over morphemes contains more complex structures and a greater number of unique features (i.e. greater $|Base|$) that a naive grammar over unsegmented words that generates the same sentences. However, this pays off in terms of other metrics: a lexicon that treats words as complex structures rather than atomic LIs can

avoid syntactic and phonological redundancies (represented by $\sum_{syn}$ and $\sum_{phon}$ respectively) by reusing lexical items in multiple words.

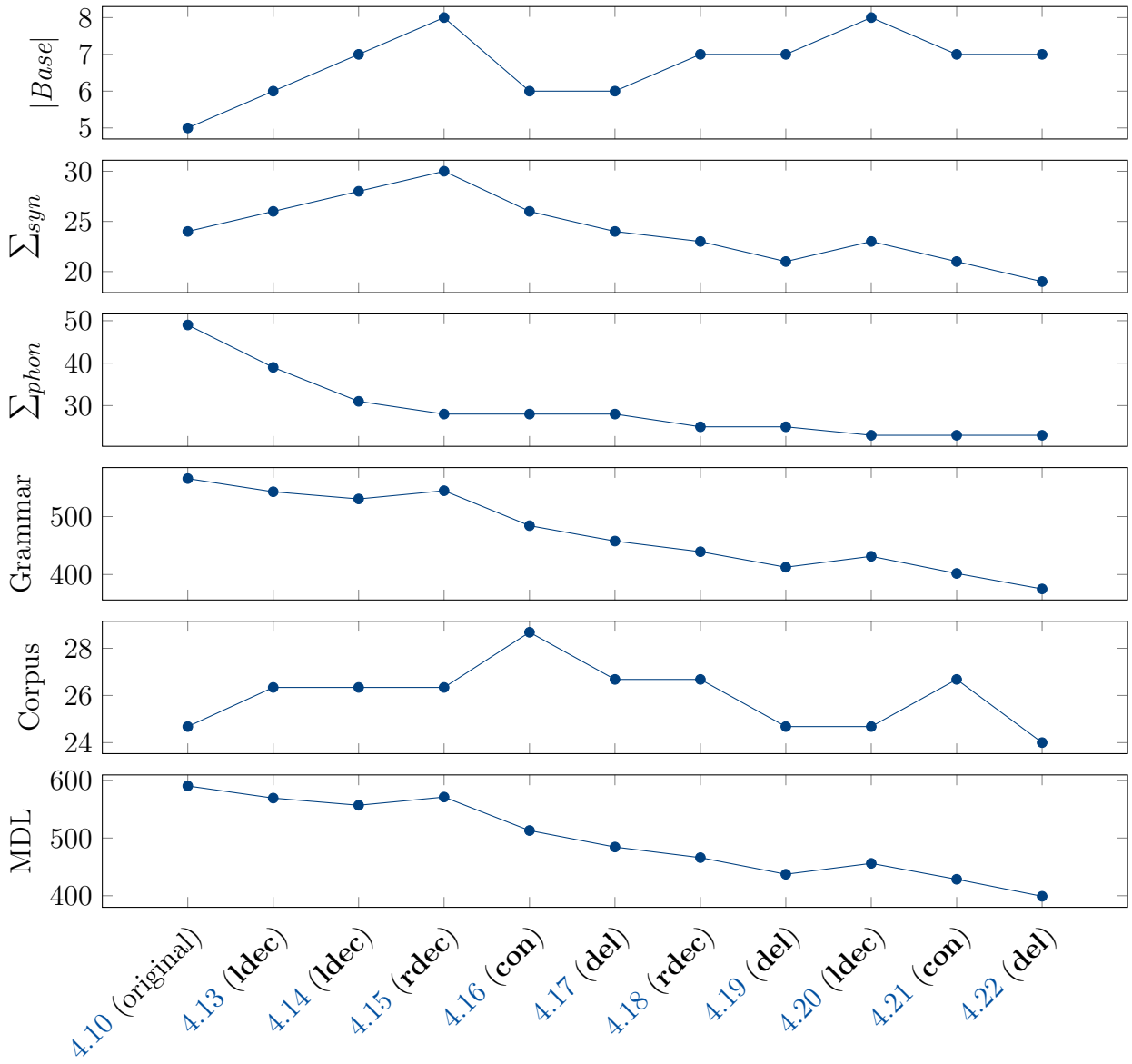


Figure 4.25: Step-by-step metrics

That said, changes undergone by these measures are not necessarily monotonic. Consider the plots across transformation steps in 4.25. The only metric that decreases monotonically is $\sum_{phon}$. Each batch decomposition step increases $|Base|$ by one and, depending on the syntactic similarity within the batch, may increase $\sum_{syn}$ as well. At the same time, decom-

position may produce category changers, feeding both edge contraction and edge deletion, which in their turn reduce $\sum_{syn}$ (and, in the case of edge contraction, also $|Base|$) without incurring any additional cost to the grammar. For example, while $\sum_{syn}$ hits a local maximum after decomposition of *be* in 4.20, this move feeds the subsequent contraction and deletion steps and ultimately allows $\sum_{syn}$ to reach the minimum in 4.22. On the other hand, the corpus cost goes up as the grammar cost falls at 4.13, 4.16, and 4.21, and does not reach the minimum of 24 bits until 4.22. The corpus cost in this example is very small compared to the grammar, so the effect on MDL is negligible. However, it offers an insight into how it can contribute to the overall MDL measure.

The following chapter identifies and addresses a number of problems inherent in the task of navigating the search space of grammars, takes a closer look at the complexity measure, and outlines an automated procedure for grammar optimization based on lexical item decomposition.

Chapter 5

Towards a learning algorithm

5.1 Overall architecture

In order to automate grammar optimization, we need a way of searching the space of candidate grammars. Even if lexical item decomposition is limited to non-trivial batches within the lexicon, the number of steps available for a single input grammar is so large that checking all possible candidate grammars is generally unfeasible. To mitigate this problem, the procedure for MG optimization developed here uses a heuristic known as *beam search* (Reddy 1977). This technique revolves around keeping track of a (relatively small) number of best hypotheses in the search space, expanding them to obtain new candidates, and discarding the rest. This parameter, *beam size*, is referred to as *bs*.

The flowchart in 5.1 provides a high-level description of the procedure. In our case, the system maintains two sets of MGs: K, which includes all grammars generated so far, and C, which contains up to *bs* candidate grammars to be processed. At each step, these candidates are used to produce new grammars via batch decomposition, edge construction, and edge deletion. Between steps, only *bs* best candidates are retained in C. Once a specified stopping condition is met, the procedure outputs the best known grammar selected from K and halts.

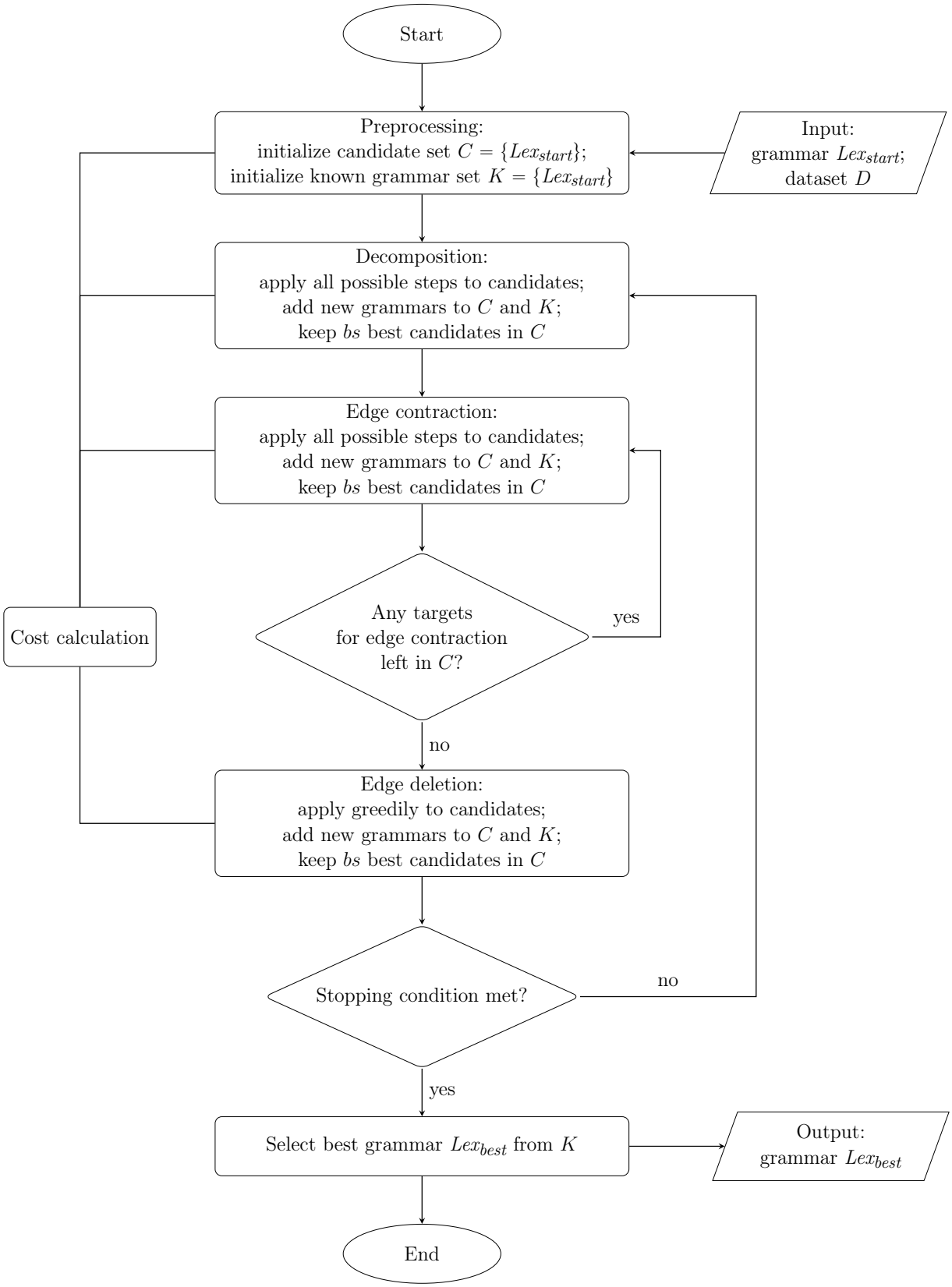


Figure 5.1: Flowchart of MG optimization

The main components of the procedure outlined in 5.1 operate as follows.

- **Preprocessing.** The input to grammar optimization consists of a single grammar Lex_{start} and a dataset D represented as a list of derivation trees generated by Lex_{start}. The procedure starts by initializing both C and K as singleton sets containing the input grammar.

- **Cost calculation.** Whenever a new grammar is constructed, the MDL-based evaluation measure is calculated and stored along with the grammar. This includes both the grammar cost and the cost of the corpus given the grammar. The latter is more problematic given the time and memory demands inherent in re-processing the entire dataset at each step; this issue is addressed in Section 5.2.

- **Decomposition.** Because there is no limit to how many decomposition steps can be performed on a given grammar, this operation proceeds one step at a time. Each candidate grammar is expanded into a set of grammars obtained from it through application of **ldec** or **rdec**. The main problem related to decomposition is selecting promising batches, which is further discussed in Section 5.3.

- **Edge contraction.** Unlike decomposition, the number of edge contraction steps is limited by how many category changers there are in a given grammar. This allows all possible steps to be performed on the same candidate before moving on to the next operation. At the same time, edge contraction has to be constrained to avoid overgeneralization; see Section 6.1 for more details. Because of this, the operation proceeds in a cycle, targeting and eliminating eligible category changers within each candidate grammar one by one and storing the newly formed grammars, until there are no targets left.

- **Edge deletion.** This operation proceeds in a greedy fashion. For each candidate grammar in C, this operation constructs a subgraph of the head-complement graph,

which only includes edges representing category changers. Any edge $\langle \texttt{x}, \texttt{y} \rangle$ is deleted and its corresponding LI ϵ :: `=>x y` is removed from the grammar as long as the graph contains another path from `x` to `y`.

- **Stopping condition.** The procedure keeps track of a specified number of best grammars in K. If these costs remain unchanged after an iteration, the procedure halts. For all experiments throughout this and next chapter, this parameter was set to 50.

A Python implementation of the optimization procedure, along with input grammars used for experiments, is available at https://github.com/mermolaeva/mg-optimizer.

5.2 Corpus encoding

5.2.1 From derivation trees to annotated CFGs

The corpus encoding scheme introduced in Chapter 3 assumes that the data is represented as a set of CFG trees (which correspond to MG derivation trees). In order to calculate the cost of a sentence, we have to traverse its derivation tree, adding up the costs of CFG rules used in it. Whenever the MG is modified (via batch decomposition, edge contraction, or edge deletion), all derivation trees that contain LIs involved in the operation change as well. In addition, since the number of CFG rules sharing the left-hand side may also become different, cost changes are not necessarily limited to sentences containing decomposed items.

Consider the left decomposition step from 4.10–4.13, repeated in 5.2, and the pair of sample CFG trees in 5.3. Before decomposition, the cost of *Mary laughs* (5.6a) and *Mary jumps* is the same, $\log_2 3 \approx 1.58$ bit encoding the choice to rewrite (`+k t, -k`) as (`=d +k t`) (`d -k`) plus $\log_2 2 = 1$ bit to encode the selection of *laughs* or *jumps*. Decomposition of the *laugh* batch adds an extra derivation step for each occurrence of *laugh*, *laughing*, or *laughs* in the corpus. After decomposition, *Mary laugh-s* (5.6b) contains only one rule with nonzero cost: (`+k t, -k`) → (`=>f1 +k t`) (`f1, -k`).

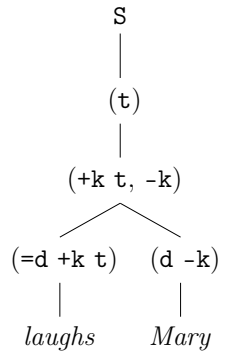
Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
laughs :: `=d +k t`
laughing :: `=d g`
laugh :: `=d v`
jumps :: `=d +k t`
jumping :: `=d g`
jump :: `=d v`

(a) Original

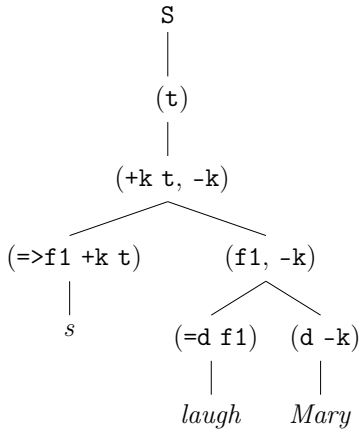
Mary :: `d -k`
bes :: `=g +k t`
wills :: `=v +k t`
be :: `=g v`
-s :: `=>f1 +k t`
-ing :: `=>f1 g`
-ϵ :: `=>f1 v`
laugh :: `=d f1`
jumps :: `=d +k t`
jumping :: `=d g`
jump :: `=d v`

(b) Decomposition of *laugh*

Figure 5.2: MGs from Chapter 4 (=4.10, 4.13)



(a) *Mary laughs* (using 5.2a)



(b) *Mary laugh-s* (using 5.2b)

Figure 5.3: CFG derivation trees: decomposition

With this new rule in place, there are now four rules with the left-hand side `(+k t, -k)`. This affects both *Mary laugh-s* and *Mary jumps*, even though the latter uses exactly the same rules as before decomposition. Each of the sentences costs $2 \log_2 2 = 2$ bits.

How can we keep track of these changes through the process of grammar optimization? A straightforward but naive approach would be to recalculate the cost of each sentence from scratch. This would involve re-parsing the entire corpus to obtain new derivation trees and

re-generating the CFG each time the MG is tweaked, as well as recording and storing the modified corpus for each candidate grammar. For larger corpora, the time and memory demands would be prohibitively high. To circumvent most of these problems, I propose an alternative approach which relies on the following three observations:

- All trees containing the same decomposed, contracted, or deleted lexical item undergo the same exact changes;
- Corpus cost calculations do not rely on structural information; it is sufficient to know which rules have been used and how many times;
- Changes to CFG rules are always local and limited to expressions headed by the LI which is being modified.

By definition, decomposition takes place before the category feature. Since each CFG rule marks an instance of MG feature checking, the number of rules affected by a single decomposition step is limited by the number of features to the left of the category in the original LI's feature bundle. Edge contraction and deletion exclusively target category changers – lexical items which make no contribution to the syntactic expression beyond changing it head's category feature. Thus, all changes to the CFG rules are confined to the initial chains of the left-hand side expression and the first expression on the right-hand side.

With this in mind, the set of derivation trees can be collapsed into a *CFG with usage data*. In such a CFG, rules are annotated with the number of occurrences associated with each LI that can head the expression on the left-hand side, as well as a number indicating which feature of the LI is checked by the derivation step associated with the rule. This concept is made precise in Definition 5.1.

Definition 5.1: CFG with usage data

Let *Lex* be a minimalist grammar, R the set of rules in the CFG generating the set of *Lex*'s derivation tree yields, and D a dataset of trees generated by R. Then the corresponding *CFG with usage data* with respect to D is the set of all triples $\langle \mathit{left}, \mathit{right}, \mathit{usage} \rangle$ such that $\langle \mathit{left}, \mathit{right} \rangle \in R$ and *usage* is the set of all triples $\langle s :: \delta, i, n \rangle$ such that:

- $s :: \delta \in \mathit{Lex}$ is a lexical item that can head a syntactic expression whose sequence of feature bundles in chain notation is *left*. Extending the MG terminology, we say that the CFG rule $\langle \mathit{left}, \mathit{right} \rangle$ can *be headed by* $s :: \delta \in \mathit{Lex}$;
- i is the index of the feature in δ that is checked by the **merge** or **move** step corresponding to the rule $\langle \mathit{left}, \mathit{right} \rangle$. For terminal rules, $i = 0$. The initial rule $\texttt{S} \rightarrow (\texttt{t})$ is considered checking of the `t` feature for this purpose;
- n is the number of times the rule $\langle \mathit{left}, \mathit{right} \rangle$ headed by $s :: \delta \in \mathit{Lex}$ occurs in D.

An annotated CFG contains less information than the original corpus and does not uniquely identify it; for instance, *Mary praises John* and *John praises Mary* would be indistinguishable by their usage data. However, it still has all information required for the task. Given a CFG with usage data G, the corpus cost is calculated as follows:

$$\sum_{\langle \mathit{left}, \mathit{right}, \mathit{usage} \rangle \in G} \Bigg(\underbrace{\log_2 \left(\left| \{ \mathit{right}' : \langle \mathit{left}, \mathit{right}', \mathit{usage}' \rangle \in G \} \right| \right)}_{\text{rule cost}} \times \underbrace{\sum_{\langle s :: \delta, i, n \rangle \in \mathit{usage}} n}_{\text{\# of occurrences}} \Bigg).$$

CFGs with usage data for 5.2a and 5.2b are given in Table 5.1 and Table 5.2, respectively. Note that only rules originally headed by *laugh*, *laughing*, or *laughs* undergo changes in the left-hand side, right-hand side, and/or usage. These rules, as well as the newly constructed terminal rules for *laugh* and *-s* and the rule representing **merge** of the latter, are highlighted in the tables with a checkmark symbol (✓). However, many unrelated rules sharing the left-hand side with them end up having a different cost.

Rule	Head	Index	Uses	Cost (bits)	
S → (t)	*bes* :: =g +k t	3	2	$8 \log_2 1 = 0$	✓
	wills :: =v +k t	3	4		
	laughs :: =d +k t	3	1		
	jumps :: =d +k t	3	1		
(t) → (+k t, -k)	*bes* :: =g +k t	2	2	$8 \log_2 1 = 0$	✓
	wills :: =v +k t	2	4		
	laughs :: =d +k t	2	1		
	jumps :: =d +k t	2	1		
(+k t, -k) → (=d +k t) (d -k)	*laughs* :: =d +k t	1	1	$2 \log_2 3 \approx 3.17$	✓
	jumps :: =d +k t	1	1		
(+k t, -k) → (=g +k t) (g, -k)	*bes* :: =g +k t	1	2	$2 \log_2 3 \approx 3.17$	
(+k t, -k) → (=v +k t) (v, -k)	*wills* :: =v +k t	1	4	$4 \log_2 3 \approx 6.34$	
(g, -k) → (=d g) (d -k)	*laughing* :: =d g	1	2	$4 \log_2 1 = 0$	✓
	jumping :: =d g	1	2		
(v, -k) → (=d v) (d -k)	*laugh* :: =d v	1	1	$2 \log_2 2 = 2$	✓
	jump :: =d v	1	1		
(v, -k) → (=g v) (g, -k)	*be* :: =g v	1	2	$2 \log_2 2 = 2$	
(d -k) → *Mary*	*Mary* :: d -k	0	8	$8 \log_2 1 = 0$	
(=g +k t) → *bes*	*bes* :: =g +k t	0	2	$2 \log_2 1 = 0$	
(=v +k t) → *wills*	*wills* :: =v +k t	0	4	$4 \log_2 1 = 0$	
(=g v) → *be*	*be* :: =g v	0	2	$2 \log_2 1 = 0$	
(=d +k t) → *laughs*	*laughs* :: =d +k t	0	1	$\log_2 2 = 1$	✓
(=d g) → *laughing*	*laughing* :: =d g	0	2	$2 \log_2 2 = 2$	✓
(=d v) → *laugh*	*laugh* :: =d v	0	1	$\log_2 2 = 1$	✓
(=d +k t) → *jumps*	*jumps* :: =d +k t	0	1	$\log_2 2 = 1$	
(=d g) → *jumping*	*jumping* :: =d g	0	2	$2 \log_2 2 = 2$	
(=d v) → *jump*	*jump* :: =d v	0	1	$\log_2 2 = 1$	
Total				≈ 24.68	

Table 5.1: CFG rule usage before decomposition (5.2a)

Rule	Head	Index	Uses	Cost (bits)	
S → (t)	*bes* :: =g +k t	3	2	$8 \log_2 1 = 0$	✓
	wills :: =v +k t	3	4		
	-s :: =>f1 +k t	3	1		
	jumps :: =d +k t	3	1		
(t) → (+k t, -k)	*bes* :: =g +k t	2	2	$8 \log_2 1 = 0$	✓
	wills :: =v +k t	2	4		
	-s :: =>f1 +k t	2	1		
	jumps :: =d +k t	2	1		
(+k t, -k) → (=>f1 +k t) (f1, -k)	*-s* :: =>f1 +k t	1	1	$\log_2 4 = 2$	✓
(+k t, -k) → (=d +k t) (d -k)	*jumps* :: =d +k t	1	1	$\log_2 4 = 2$	
(+k t, -k) → (=g +k t) (g, -k)	*bes* :: =g +k t	1	2	$2 \log_2 4 = 4$	
(+k t, -k) → (=v +k t) (v, -k)	*wills* :: =v +k t	1	4	$4 \log_2 4 = 8$	
(f1, -k) → (=d f1) (d -k)	*laugh* :: =d f1	1	4	$4 \log_2 1 = 0$	✓
(g, -k) → (=>f1 g) (f1, -k)	*-ing* :: =>f1 g	1	2	$2 \log_2 2 = 2$	✓
(g, -k) → (=d g) (d -k)	*jumping* :: =d g	1	2	$2 \log_2 2 = 2$	
(v, -k) → (=>f1 v) (f1, -k)	*-ϵ* :: =>f1 v	1	1	$1 \log_2 3 \approx 1.58$	✓
(v, -k) → (=d v) (d -k)	*jump* :: =d v	1	1	$1 \log_2 3 \approx 1.58$	
(v, -k) → (=g v) (g, -k)	*be* :: =g v	1	2	$2 \log_2 3 = \approx 3.17$	
(d -k) → *Mary*	*Mary* :: d -k	0	8	$8 \log_2 1 = 0$	
(=g +k t) → *bes*	*bes* :: =g +k t	0	2	$2 \log_2 1 = 0$	
(=v +k t) → *wills*	*wills* :: =v +k t	0	4	$4 \log_2 1 = 0$	
(=g v) → *be*	*be* :: =g v	0	2	$2 \log_2 1 = 0$	
(=d f1) → *laugh*	*laugh* :: =d f1	0	4	$4 \log_2 1 = 0$	✓
(=>f1 +k t) → *-s*	*-s* :: =>f1 +k t	0	1	$\log_2 1 = 0$	✓
(=>f1 g) → *-ing*	*-ing* :: =>f1 g	0	2	$2 \log_2 1 = 0$	✓
(=>f1 v) → *-ϵ*	*-ϵ* :: =>f1 v	0	1	$1 \log_2 1 = 0$	✓
(=d +k t) → *jumps*	*jumps* :: =d +k t	0	1	$\log_2 1 = 0$	
(=d g) → *jumping*	*jumping* :: =d g	0	2	$2 \log_2 2 = 0$	
(=d v) → *jump*	*jump* :: =d v	0	1	$\log_2 1 = 0$	
Total				≈ 26.34	

Table 5.2: CFG rule usage after decomposition (5.2b)

The original dataset of derivation trees D is converted into a CFG with usage data as part of the preprocessing component. From a high-level viewpoint, whenever an LI is modified, the algorithm obtains the new corpus cost from the annotated CFG by performing the following steps:

- Identify which CFG rules are associated with this LI, and how many times they were used throughout the original corpus;
- Construct new CFG rules reflecting the change to the original MG;
- Reassign usage data associated with the LI from the original rules to the new rules; delete any rules that no longer have any usage data;
- Recalculate the corpus cost with respect to the new CFG.

In what follows, I outline in more detail the algorithms modifying the CFG with usage data in response to lexical item decomposition, edge contraction, and edge deletion. For simplicity, all algorithms are defined for a single LI rather than a set or batch.

5.2.2 Decomposition of lexical items

During lexical item decomposition, a single original LI is split into two new items: the lower LI, carrying a fresh category feature, and the upper LI, which selects the former. Each rule that can be headed by the original LI is modified depending on which of the newly formed LIs becomes its new head. This operation is carried out by `CFG-Decompose` of Algorithm 5.1. It assumes an additional auxiliary function `CFG-Add`, which takes as input a CFG with usage data and a rule and updates the rule's usage data if it is already present in the grammar, or adds the rule to the grammar if it is not.

Function `CFG-Decompose`$(G, i, s_{old} :: \delta_{old}, s_{upper} :: \delta_{upper}, s_{lower} :: \delta_{lower})$

Data: CFG with usage data G, index i, old LI $s_{old} :: \delta_{old}$, new LIs $s_{upper} :: \delta_{upper}, s_{lower} :: \delta_{lower}$

Result: CFG with usage data G'

Initialize $G' = \varnothing$

foreach $\langle left, right, usage \rangle$ in G **do**

  if $\langle s_{old} :: \delta_{old}, j, n \rangle$ in $usage$, where $j, n \in \mathbb{N}$ **then**

    if $j > i$ **then** `// headed by the upper LI`

      $usage' = usage[\langle s_{old} :: \delta_{old}, j, n \rangle \mapsto \langle s_{upper} :: \delta_{upper}, j - i + 1, n \rangle]$

      $G' =$ `CFG-Add`$(G', \langle left, right, usage' \rangle)$

    else `// headed by the lower LI`

      Let $left'$ be identical to $left$, except for the first feature bundle replaced with $\delta_{lower}[j + 1, |\delta_{lower}|]$

      if $\langle left, right, usage \rangle$ is a terminal rule **then**

        $right' = s_{lower}$ `// lower terminal rule`

      else

        Let $right'$ be identical to $right$, except for the first feature bundle of the first nonterminal replaced with $\delta_{lower}[j, |\delta_{lower}|]$

      $G' =$ `CFG-Add`$(G', \langle left', right', \{\langle s_{lower} :: \delta_{lower}, j, n \rangle\} \rangle)$

      $G' =$ `CFG-Add`$(G', \langle left, right, usage - \{\langle s_{old} :: \delta_{old}, j, n \rangle\} \rangle)$

      if $j = i$ **then** `// merge and upper terminal rules`

        $G' =$ `CFG-Add`$(G', \langle left, \langle \delta_{upper}, left' \rangle, \{\langle s_{upper} :: \delta_{upper}, 1, n \rangle\} \rangle)$

        $G' =$ `CFG-Add`$(G', \langle \delta_{upper}, s_{upper}, \{\langle s_{upper} :: \delta_{upper}, 0, n \rangle\} \rangle)$

  else `// unrelated rule`

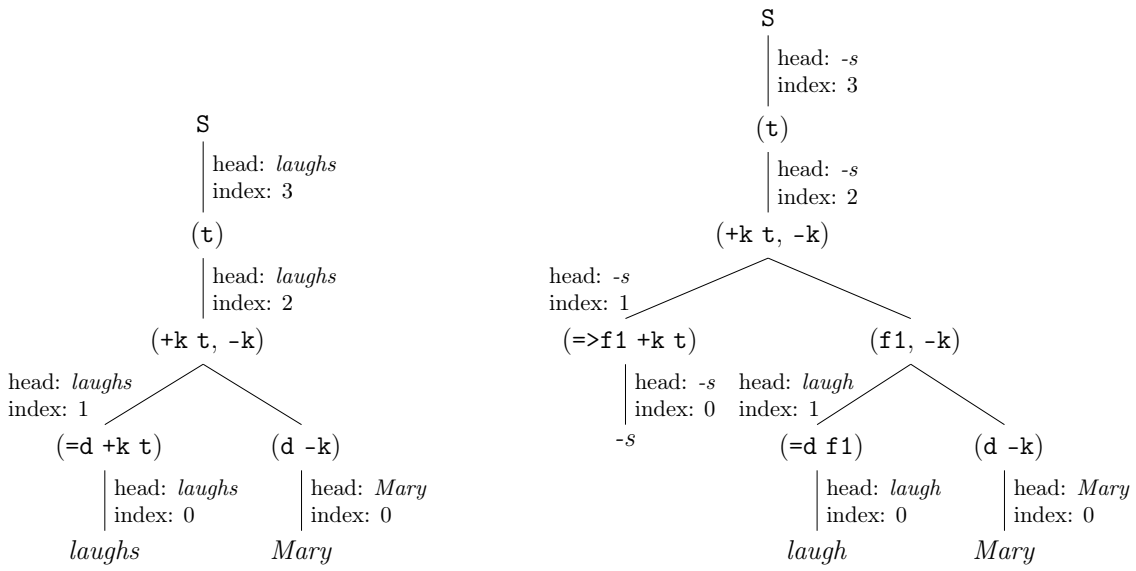
    $G' =$ `CFG-Add`$(G', \langle left, right, usage \rangle)$

return G'

Algorithm 5.1: Changing a CFG in response to decomposition

The function determines how to process each rule by comparing the index i at which the original LI was decomposed to the index j of the checked feature stored as part of the usage data triple. In the case of left decomposition, $i = l_{syn}$. In the case of right decomposition, for each LI $s :: \delta$ in the batch, $i = |\delta| - l_{syn}$. If $j > i$, the rule is headed by the upper LI. Both the left-hand side and the right-hand side remain unchanged, whereas the usage data is tweaked to reflect the new index and head LI. If $j \leqslant i$, the algorithm constructs a modified rule headed by the lower LI, modifying both the left-hand side and right-hand side. In addition, if $j = i$, a new rule is formed to represent the **merge** step combining the upper LI with the expression headed by the lower LI. Finally, terminal rules for both new LIs are added to the grammar.

Consider again the decomposition example from the previous section. The CFG trees are repeated once more in 5.4, annotated with head and index information.



(a) *Mary laughs* (using Table 5.1) (b) *Mary laugh-s* (using Table 5.2)

Figure 5.4: CFG derivation trees (=5.3) annotated with usage data

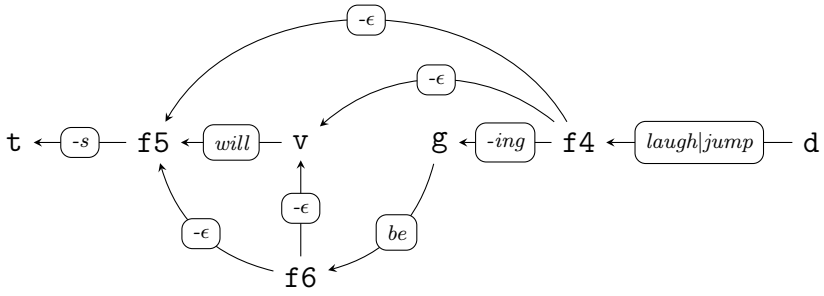
The old LI *laughs* :: `=d +k t` is decomposed at index $i = 1$. The lower LI is *laugh* :: `=d f1`, and the upper LI is *-s* :: `=>f1 +k t`. Then `CFG-Decompose` proceeds as follows.

- The usage of `S` → (`t`) includes the old LI (index $j = 3$). In G', this is replaced with the upper LI (index $j - i + 1 = 3$), and the rest of the rule stays the same;
- The usage of (`t`) → (`+k t`, `-k`) includes the old LI (index $j = 2$). In G', this is replaced with the upper LI (index $j - i + 1 = 2$), and the rest of the rule stays the same;
- The usage of (`+k t`, `-k`) → (`=d +k t`) (`d -k`) includes the old LI (index $j = 1$).
  - The original rule is added to G' with the old LI removed from its usage data;
  - A new rule headed by the lower LI is added to G': (`f1`, `-k`) → (`=d f1`) (`d -k`) (index $j = 1$). Its left-hand side is based on $\delta_{lower}[j + 1, |\delta_{lower}|] = \delta_{lower}[2, 2] =$ `f1`, and its right-hand side is based on $\delta_{lower}[j, |\delta_{lower}|] = \delta_{lower}[1, 2] =$ `=d f1`;
  - Since $j = i = 1$, the following new rule headed by the upper LI represents the **merge** step: (`+k t`, `-k`) → (`=>f1 +k t`) (`f1`, `-k`) (index 1);
  - Finally, we add a terminal rule for the upper LI: (`=>f1 +k t`) → *-s* (index 0);
- The usage of (`=d +k t`) → *laughs* includes the old LI (index $j = 0$). Since this is a terminal rule, we add a terminal rule for the lower LI to G': (`=d f1`) → *laugh* (index $j - i + 1 = 0$). Its left-hand side is based on $\delta_{lower}[j + 1, |\delta_{lower}|] = \delta_{lower}[1, 2] =$ `=d f1`.
- All other rules, which cannot be headed by the old LI, are irrelevant to this decomposition step. They are added to G' unchanged.

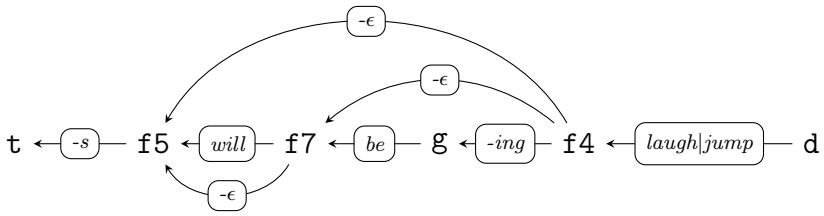
5.2.3 Auxiliary operations

The auxiliary operations of edge contraction and edge deletion apply to category changers. Since these LIs have exactly two syntactic features, a morphological selector and a category, the only rules headed by a given category changer are those that **merge** it with its complement. For example, the edge contraction step in 5.5 gets rid of the LI *-ϵ* :: `=>f6 v`, replacing all occurrences of both `f6` and `v` with a new feature `f7`. The only change for affected structures such as 5.6 is that any LI that originally selected an expression of category `v` headed

by the category changer will now instead select its complement (formerly of category `f6`) directly.

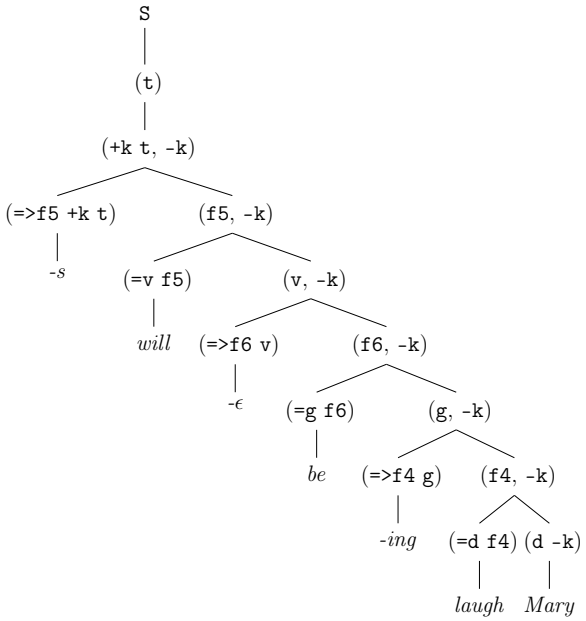


(a) Before

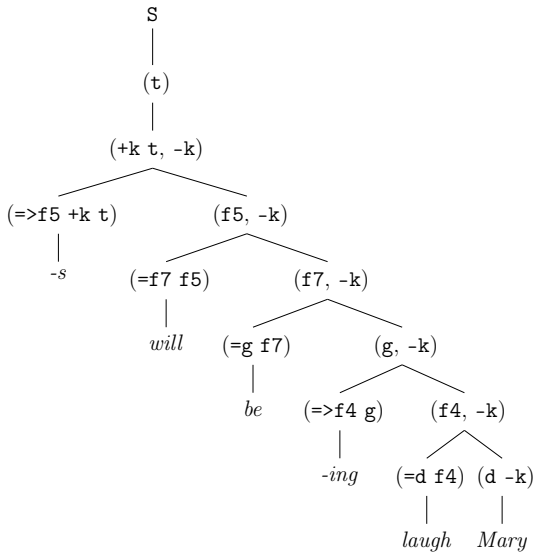


(b) After

Figure 5.5: Contraction of `f6` → `v` (=4.20, 4.21)



(a) *Mary will-s be laughing* (using 5.5a)



(b) *Mary will-s be laughing* (using 5.5b)

Figure 5.6: CFG derivation trees: edge contraction

Algorithm 5.2 exploits this observation to deal with LIs eliminated through edge contraction in a straightforward way. The function `CFG-Contract` takes as input a CFG with usage data, an LI to be contracted -ϵ :: =>$\mathtt{f}_1$ $\mathtt{f}_2$, and a fresh category feature $\mathtt{f}_{\mathtt{new}}$. For each rule, all occurrences of $\mathtt{f}_1$ and $\mathtt{f}_2$ are replaced with $\mathtt{f}_{\mathtt{new}}$; and any usage triple associated with -ϵ :: =>$\mathtt{f}_1$ $\mathtt{f}_2$ is removed from its usage data. The modified rule is added to the new grammar as long as its usage data is not empty. Finally, the algorithm generates any missing CFG rules by applying **merge** and **move** to the existing expressions in a process identical to conversion of MGs to CFGs, informally described in Subsection 2.3.4. For any new rules, possible heads are identified and assigned 0 occurrences. This is represented in the pseudocode as an auxiliary function `CFG-Closure`.

Function `CFG-Contract`(G, -ϵ :: =>$\mathtt{f}_1$ $\mathtt{f}_2$, $\mathtt{f}_{\mathtt{new}}$)
  Data: CFG with usage data G, old LI -ϵ :: =>$\mathtt{f}_1$ $\mathtt{f}_2$, new category feature $\mathtt{f}_{\mathtt{new}}$
  Result: CFG with usage data G'
  Initialize $G' = \varnothing$
  foreach $\langle left, right, usage \rangle$ in G **do**
    $usage' = usage - \{\langle$-$\epsilon$ :: =>$\mathtt{f}_1$ $\mathtt{f}_2, j, n\rangle$ for any $j, n \in \mathbb{N}\}$
    if $usage' \neq \varnothing$ **then**
      $left' = left[\mathtt{f}_1 \mapsto \mathtt{f}_{\mathtt{new}}, \mathtt{f}_2 \mapsto \mathtt{f}_{\mathtt{new}}]$
      $right' = right[\mathtt{f}_1 \mapsto \mathtt{f}_{\mathtt{new}}, \mathtt{f}_2 \mapsto \mathtt{f}_{\mathtt{new}}]$
      $G' =$ `CFG-Add`$(G', \langle left', right', usage' \rangle)$
  return `CFG-Closure`(G')

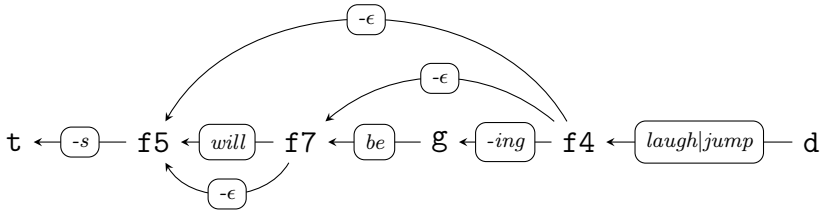
Algorithm 5.2: Changing a CFG in response to edge contraction

To provide a concrete illustration, `CFG-contract` makes the following changes to CFG rules involved in 5.6:

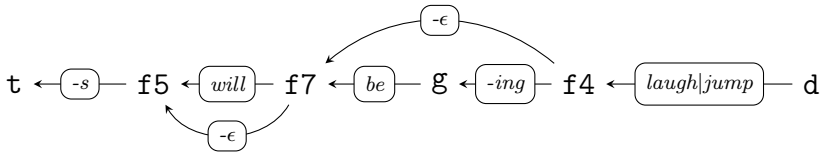
- The rule (`f5`, `-k`) → (`=v f5`) (`v`, `-k`) cannot be headed by the old LI. Its usage remains unchanged, and the rule itself changes to (`f5`, `-k`) → (`=f7 f5`) (`f7`, `-k`);

- The rule (v, -k) → (=>f6 v) (f6, -k) can only be headed by the old LI. Since this usage triple is removed, the entire rule is not added to G';
- The terminal rule (=>f6 v) → -ϵ can only be headed by the old LI. Since this usage triple is removed, the entire rule is not added to G';
- The rule (f6, -k) → (=g f6) (g, -k) cannot be headed by the old LI. Its usage remains unchanged, and the rule itself changes to (f7, -k) → (=g f7) (g, -k);
- None of the other rules in G can be headed by the old LI. They are added to G' unchanged, except for all occurrences of f6 and v, which are replaced by f7.

Finally, edge deletion replaces a category changer with a sequence of other category changers already present in the minimalist grammar. A nontrivial edge deletion step (also discussed earlier in Chapter 4) is shown in 5.7. Before the operation is applied (5.7a), -ϵ :: =>f4 f5 occurs twice in the corpus – in the derivation of *Mary laugh-s* (shown in 5.8a) and *Mary jump-s*. Similarly, -ϵ :: =>f4 f7 and -ϵ :: =>f7 f5 are used two times each – in *Mary will-s laugh* | *jump* and *Mary be-s laughing* | *jumping*, respectively. Edge deletion (5.7b, 5.8b) removes -ϵ :: =>f4 f5 from the grammar. Both sentences in the dataset that contain it are restructured to use -ϵ :: =>f4 f7 and -ϵ :: =>f7 f5 instead.

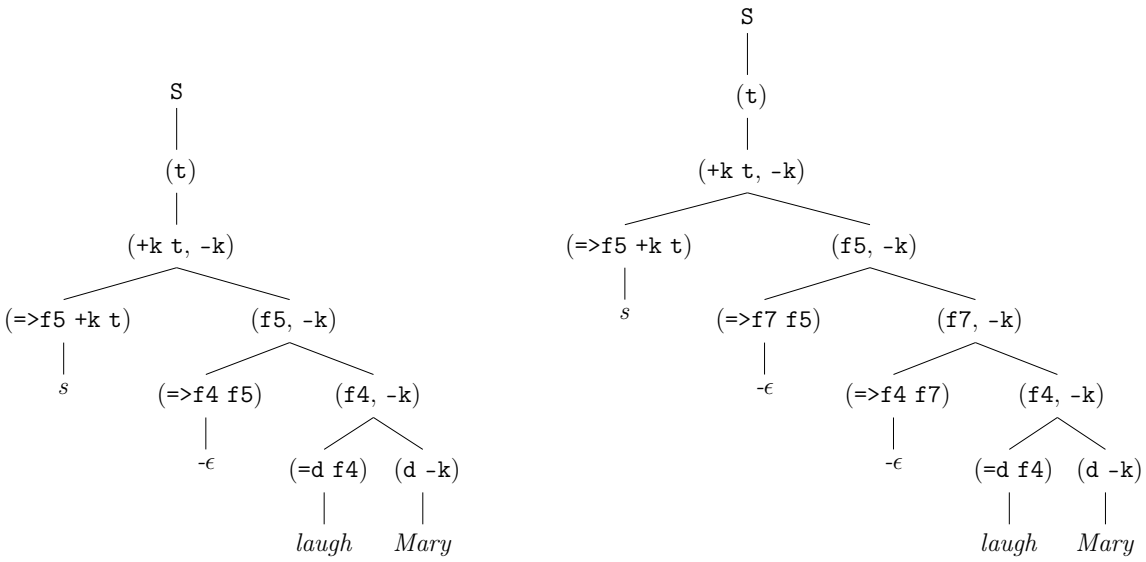


(a) Before



(b) After

Figure 5.7: Deletion of f4 → f5 (=4.21, 4.22)



(a) *Mary laugh-s* (using 5.7a) (b) *Mary laugh-s* (using 5.7b)

Figure 5.8: CFG derivation trees: edge deletion

With respect to CFG rules, edge deletion is similar to lexical item decomposition in that it replaces each rule associated with a single LI with those encoding multiple other LIs. Unlike decomposition, rather than create new CFG rules, it reassigns occurrences to existing rules. This intuition is formalized in Algorithm 5.3. To reassign occurrences of the **merge** rules involving the old LI -ϵ :: =>$\mathtt{f}_1$ $\mathtt{f}_m$, the function `CFG-Delete-Edge` iterates over the new category changers -ϵ :: =>$\mathtt{f}_1$ $\mathtt{f}_2$, ..., -ϵ :: =>$\mathtt{f}_{n-1}$ $\mathtt{f}_m$. For each -ϵ :: =>$\mathtt{f}_i$ $\mathtt{f}_{i+1}$, it identifies the relevant rule. Its left-hand side is the same as the original **merge** rule, except for the first feature replaced by $\mathtt{f}_i$. On the right-hand side, the first nonterminal symbol is the new LI in question, and the second nonterminal symbol is the left-hand side of the previous rule. Similarly, all occurrences of the terminal rule introducing -ϵ :: =>$\mathtt{f}_1$ $\mathtt{f}_m$ are removed, and the same number of occurrences is added to each of the terminal rules introducing the new LIs.

Function `CFG-Delete-Edge`$(G, \text{-}\epsilon$ :: `=>f1 fm`$, \{\text{-}\epsilon$ :: `=>f1 f2`$, ..., \text{-}\epsilon$ :: `=>fn−1 fm`$\})$

Data: CFG with usage data G, old LI -ϵ :: `=>f1 fm`, new LIs $\{\text{-}\epsilon$ :: `=>f1 f2`$, ..., \text{-}\epsilon$ :: `=>fn−1 fm`$\}$

Result: CFG with usage data G'

Initialize $G' = \varnothing$

foreach $\langle left, right, usage \rangle$ in G **do**

  if $\langle \text{-}\epsilon$ :: `=>f1 fm`$, j, n \rangle$ in $usage$, where $j, n \in \mathbb{N}$ **then**

    $usage' = usage - \{\langle \text{-}\epsilon$ :: `=>f1 fm`$, j, n \rangle\}$

    if $\langle left, right, usage \rangle$ is a nonterminal rule **then**

      $right_2 = right[2]$

      foreach $i \in [1, m-1]$ **do** `// merging in the new LIs`

        Let $left'$ be identical to $left$, except for the first feature replaced with `fi+1`

        $right' =$ (`=>fi fi+1`) ($right_2$)

        $G' =$ `CFG-Add`$(G', \langle left', right', \{\text{-}\epsilon$ :: `=>fi fi+1`$, j, n\} \rangle)$

        $right_2 = left'$

      if $usage' \neq \varnothing$ **then**

        $G' =$ `CFG-Add`$(G', \langle left, right, usage' \rangle)$

    else `// terminal rule for the old LI`

      foreach $i \in [1, m-1]$ **do** `// terminal rules for the new LIs`

        $G' =$ `CFG-Add`$(G', \langle$ (`=>fi fi+1`)$, \text{-}\epsilon, \{\langle \text{-}\epsilon$ :: `=>fi fi+1`$, j, n \rangle\} \rangle)$

  else `// unrelated rule`

    $G' =$ `CFG-Add`$(G', \langle left, right, usage \rangle)$

  return G'

Algorithm 5.3: Changing a CFG in response to edge deletion

In the example 5.8, the old LI is -ϵ :: `=>f4 f5`, and the new LIs are -ϵ :: `=>f4 f7` and -ϵ :: `=>f7 f5`. `CFG-Delete-Edge` processes the associated CFG rules as follows:

- The nonterminal rule (f5, -k) → (=>f4 f5) (f4, -k) has 2 occurrences, both headed by the old LI. This rule is removed, and:
  - The rule (f7, -k) → (=>f4 f7) (f4, -k) is added to G' with 2 occurrences headed by -ϵ :: =>f4 f7;
  - The rule (f5, -k) → (=>f7 f5) (f7, -k) is added to G' with 2 occurrences headed by -ϵ :: =>f7 f5;
- The rule (=>f4 f5) → -ϵ is terminal and headed by the old LI. We add 2 occurrences to each of the following terminal rules: (=>f4 f7) → -ϵ and (=>f7 f5) → -ϵ;
- All other rules, which cannot be headed by the old LI, are added to G' with their original usage data. If a rule is already in G, the occurrences are added together.

5.3 Batches and metabatches

A major issue related to automating lexical item decomposition is batch selection. This is the problem of how to efficiently determine which words in the lexicon likely have a shared morpheme, taking into account both syntactic and phonological similarities. Our definitions of left and right batch decomposition introduced in Section 4.1 don't put many restrictions on what subsets of a lexicon are valid batches, so simply attempting all possible decomposition steps is not a viable option.

Consider the lexicon in 5.9.[18] Left decomposition does not require the LIs within the batch to have any syntactic features other than the =>x selector, if present. By the definition of **ldec** (Definition 4.1), any subset of the lexicon in 5.9 is a valid batch, producing a number of options equal to the size of the powerset of the lexicon, $|\mathcal{P}(Lex)| = 2^{|Lex|}$. Even

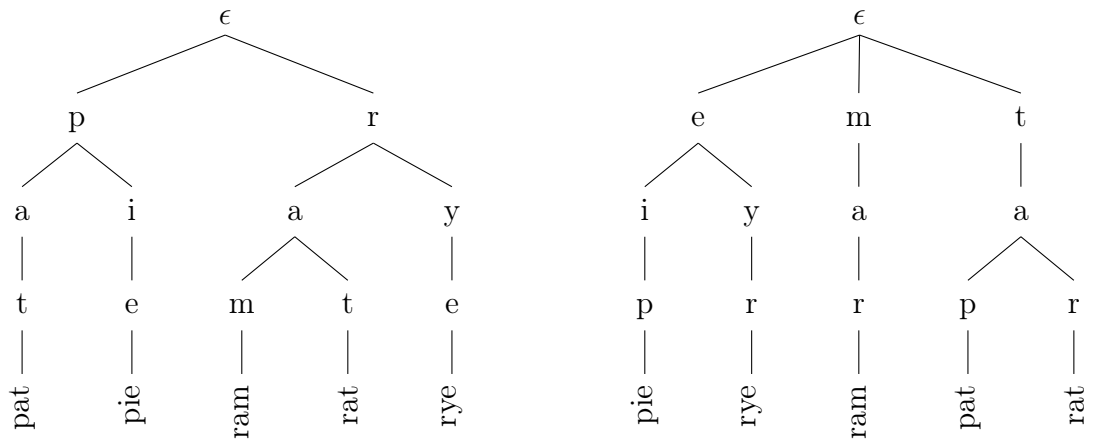
18. In this and subsequent examples, wherever possible, arguments are **merge**d on the right and later **move**d (if necessary) to achieve the correct linear order. Introducing arguments of the same category with different features (e.g. =d and =d) would effectively give the learner additional distinctions (e.g. internal vs. external DP arguments) to draw on. While the effects of this information may be interesting in their own right, for now we avoid them in the interest of keeping the input grammar as theory-neutral as possible.

considering only subsets of size two and greater, this lexicon of thirty LIs contains $(2^{30}-31) = 1,073,741,793$ potential batches. Only a few of these options are linguistically reasonable or worth exploring. Because of this, we need a way of selecting promising batches: sets of LIs which have some phonological and/or syntactic similarity between them.

Mary :: `d -k`	*dance* :: `=d v`	*hug* :: `=d *k =d v`
willd :: `=v +k t`	*danced* :: `=d +k t`	*hugd* :: `=d *k =d +k t`
wills :: `=v +k t`	*danceing* :: `=d prog`	*huging* :: `=d *k =d prog`
have :: `=perf v`	*dancen* :: `=d perf`	*hugn* :: `=d *k =d perf`
haved :: `=perf +k t`	*dances* :: `=d +k t`	*hugs* :: `=d *k =d +k t`
haves :: `=perf +k t`	*laugh* :: `=d v`	*praise* :: `=d *k =d v`
be :: `=prog v`	*laughd* :: `=d +k t`	*praised* :: `=d *k =d +k t`
bed :: `=prog +k t`	*laughing* :: `=d prog`	*praiseing* :: `=d *k =d prog`
ben :: `=prog perf`	*laughn* :: `=d perf`	*praisen* :: `=d *k =d perf`
bes :: `=prog +k t`	*laughs* :: `=d +k t`	*praises* :: `=d *k =d +k t`

Figure 5.9: English auxiliaries and lexical verbs

One way to home in on redundancies within a lexicon is to use a type of data structure known as a *trie* (Fredkin 1960). A (prefix) trie is a tree structure for storing a set of strings drawn from some alphabet. It contains one internal node for every prefix of any of the strings, with the root node labeled with ϵ and other internal nodes storing individual symbols from the alphabet. Leaf nodes store the strings themselves. For any subset of strings, the leaves that contain them are dominated by all nodes corresponding to their common prefixes. Similarly, one can construct a suffix trie whose nodes contain suffixes of strings instead of prefixes. For example, the set of strings {*pat*, *pie*, *ram*, *rat*, *rye*} would produce the prefix and suffix tries in 5.10a and 5.10b, respectively.



(a) Prefix trie (b) Suffix trie

Figure 5.10: Tries storing {*pat*, *pie*, *ram*, *rat*, *rye*}

Each minimalist lexical item is a pair of strings, one drawn from Σ and one from *Syn*. Thus, a set of lexical items can be represented as a trie with respect to either syntax (5.11) or phonology (5.12), with leaves containing LIs. Prefix tries are useful for left decomposition, while right decomposition relies on suffix tries.

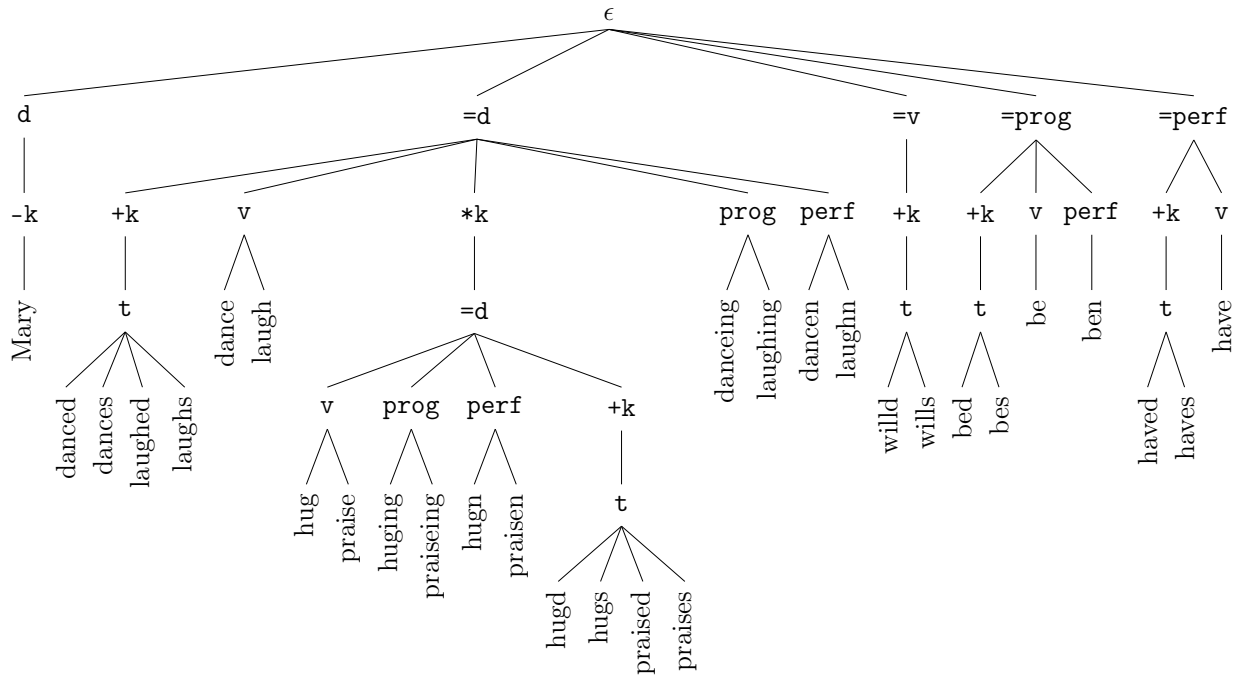


Figure 5.11: Lexicon 5.9 as a prefix trie w.r.t. syntax

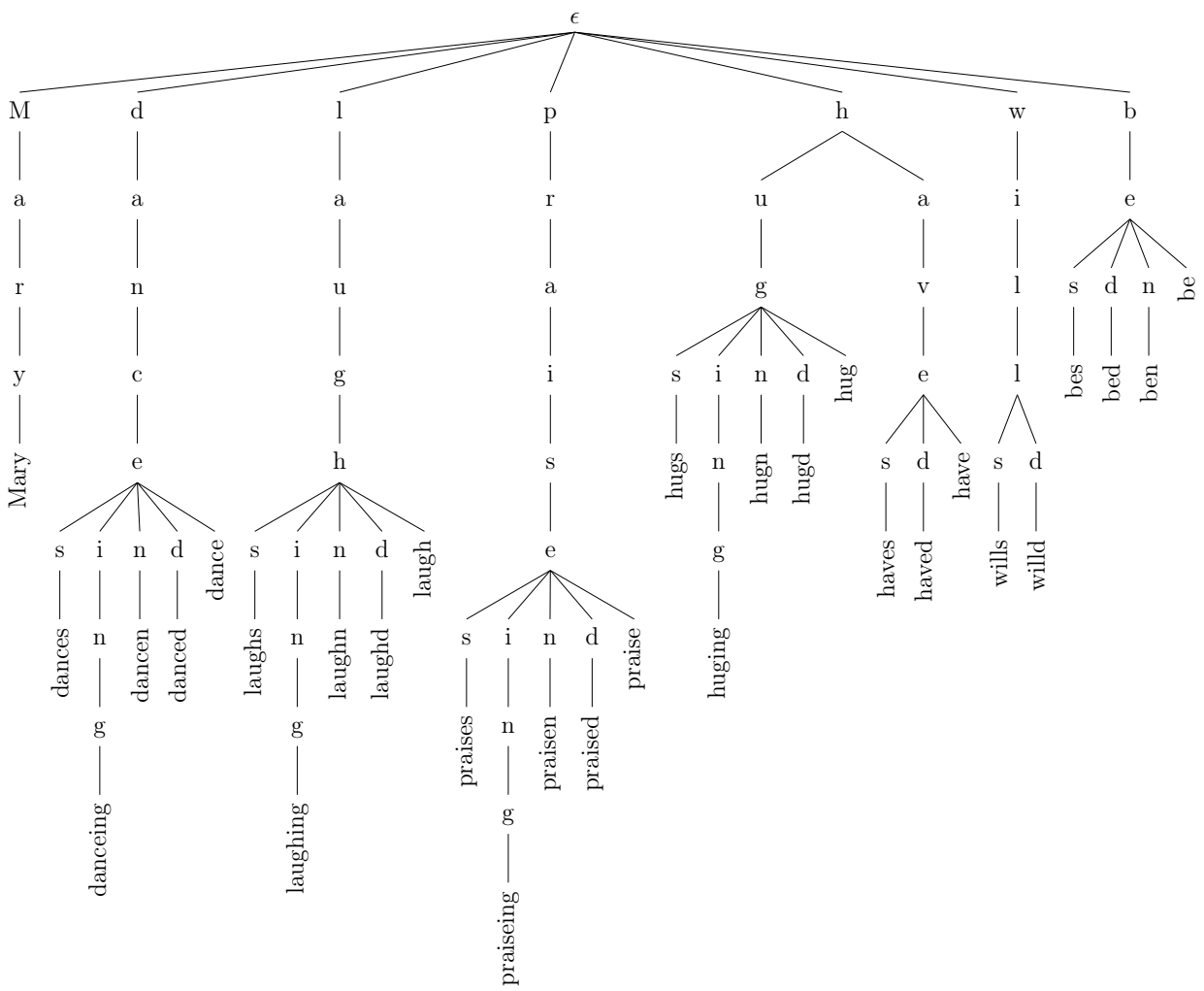


Figure 5.12: Lexicon 5.9 as a prefix trie w.r.t. phonology

Each internal node t in a syntactic prefix trie (such as 5.11) stands for a sequence of syntactic features shared by all leaves dominated by it. The feature sequence can be split anywhere as long as it does not produce unusable LIs. Using tree terms, t must not strictly dominate any node labeled with a morphological selector or be dominated by a category node. Thus, internal nodes define subsets of LIs sharing a part of their syntactic feature bundle. Similarly, phonological prefix tries (5.12) define subsets of LIs whose string components have a common prefix. To identify promising batches, we first obtain subsets of lexical items with phonological and syntactic similarities independently, and then compare them pairwise to

find lexical items present in both. For instance, prefix tries in 5.11 and 5.12 yield the batches listed in Table 5.3.

#	Batch	Shared syntax	Shared phonology
1	*be*, *bed*, *ben*, *bes*	`=prog`	*be-*
2	*bed*, *bes*	`=prog +k`	*be-*
3	*dance*, *danced*, *danceing*, *dancen*, *dances*	`=d`	*dance-*
4	*dance*, *danced*, *danceing*, *dancen*, *dances*, *hug*, *hugd*, *huging*, *hugn*, *hugs*, *laugh*, *laughd*, *laughing*, *laughn*, *laughs*, *praise*, *praised*, *praiseing*, *praisen*, *praises*	`=d`	ϵ
5	*danced*, *dances*	`=d +k`	*dance-*
6	*danced*, *dances*, *laughd*, *laughs*	`=d +k`	ϵ
7	*have*, *haved*, *haves*	`=perf`	*have-*
8	*have*, *haved*, *haves*, *hug*, *hugd*, *huging*, *hugn*, *hugs*	ϵ	*h-*
9	*haved*, *haves*	`=perf +k`	*have-*
10	*hug*, *hugd*, *huging*, *hugn*, *hugs*	`=d *k =d`	*hug-*
11	*hug*, *hugd*, *huging*, *hugn*, *hugs*, *praise*, *praised*, *praiseing*, *praisen*, *praises*	`=d *k =d`	ϵ
12	*hugd*, *hugs*	`=d *k =d +k`	*hug-*
13	*hugd*, *hugs*, *praised*, *praises*	`=d *k =d +k`	ϵ
14	*laugh*, *laughd*, *laughing*, *laughn*, *laughs*	`=d`	*laugh-*
15	*laughd*, *laughs*	`=d +k`	*laugh-*
16	*praise*, *praised*, *praiseing*, *praisen*, *praises*	`=d *k =d`	*praise-*
17	*praised*, *praises*	`=d *k =d +k`	*praise-*
18	*willd*, *wills*	`=v +k`	*will-*

Table 5.3: Batches obtained from 5.11 and 5.12

Many batches in Table 5.3 represent full paradigms of lexical verbs (##3, 10, 14, 16) or auxiliary verbs (##1, 7, 18), which are precisely what left decomposition is expected to

target. While some batches group multiple paradigms together (e.g. #8) or include only a part of a paradigm (e.g. #2), the number of obviously bad moves is relatively small.

Algorithm 5.4 formalizes the first step of this process for left decomposition. The function `Prefix-Phon` collects phonological batches of size two and greater from a trie. `Prefix-Syn` works in a similar manner for syntactic batches, ensuring a valid split of the feature sequence.

Function `Prefix-Phon`(L)
    Data: Set of lexical items L
    Result: Set of phonological batches B_{phon}
    Let T_{phon} be the phonological prefix trie of L
    Initialize $B_{phon} = \varnothing$
    foreach node t of T_{phon} **do**
        if t has more than one child **then**
            Add $\{c : c$ is a leaf dominated by $t\}$ to B_{phon}
    return B_{phon}

Function `Prefix-Syn`(L)
    Data: Set of lexical items L
    Result: Set of syntactic batches B_{syn}
    Let T_{syn} be the syntactic prefix trie of Lex
    Initialize $B_{syn} = \varnothing$
    foreach node t of T_{syn} **do**
        if t has more than one child **and**
        t does not strictly dominate any node labeled `=>x`, `x` $\in$ *Base* **and**
        t is not dominated by any node labeled `x`, `x` $\in$ *Base* **then**
            Add $\{c : c$ is a leaf dominated by $t\}$ to B_{syn}
    return B_{syn}

Algorithm 5.4: Obtaining phonological and syntactic batches from trees

The second step is represented by Algorithm 5.5. The final set of batches is obtained via `Make-Batches-Left` by considering the cross-product of phonological and syntactic batches and taking the intersection of each pair of sets.

1 **Function** `Make-Batches-Left`(Lex)
  Data: Lexicon Lex
  Result: Set of batches B
2 Initialize $B = \varnothing$
3 B_{syn} = `Prefix-Syn`(Lex)
4 B_{phon} = `Prefix-Phon`(Lex)
5 **foreach** b_i in B_{syn} **do**
6 **foreach** b_j in B_{phon} **do**
7 Add $b_i \cap b_j$ to B
8 **return** B

Algorithm 5.5: Batch formation

In its turn, Algorithm 5.6 performs left decomposition for each $b = \{s_1 :: \delta_1, ..., s_n :: \delta_n\}$ in the set of batches produced by `Make-Batches-Left`. On the phonological side, it proceeds in a greedy fashion, assigning the longest common prefix of $\{s_1, ..., s_n\}$ as the string component to the newly formed shared lexical item. On the syntactic side, it explores all ways of splitting $\{\delta_1, ..., \delta_n\}$ that comply with restrictions on **ldec**. This algorithm assumes an auxiliary function `Syn-Range`, which enforces all restrictions on lexical item decomposition and supplies the range of valid decomposition indices (values of l_{syn}) for a given set of syntactic feature bundles. New grammars resulting from decomposition are added to the set of candidates C and the set of known grammars K.

Function `Decompose-Batches-Left`(C, K, Lex)

  Data: Set of candidates C, set of known grammars K, lexicon Lex

  Result: Updated C and K

  $B =$ `Make-Batches-Left`(Lex)

  foreach $b = \{s_1 :: \delta_1, ..., s_n :: \delta_n\}$ in B **do**

    Let l_{phon} be the length of the longest common prefix of $\{s_1, ...s_n\}$

    foreach l_{syn} in `Syn-Range`$(\{\delta_1, ...\delta_n\})$ **do**

      $Lex_b =$ **ldec**$(Lex, b, l_{phon}, l_{syn})$

      Add Lex_b to C

      Add Lex_b to K

  return C, K

Algorithm 5.6: Left decomposition over batches

Returning to the lexicon in 5.9, using tries has brought the number of options considered for decomposition down to the much more manageable eighteen batches, each of which is a set of lexical items all sharing some phonological and/or syntactic properties.

This, however, uncovers a problem with our strategy of making a single decomposition step at a time. Even if multiple good moves are available, like in Table 5.3, only one can be picked at a time for a given candidate grammar. If the input lexicon contains multiple paradigms of open-class LIs (such as lexical verbs), it will take multiple steps to decompose all of them, even if all required moves are already available at the first step. For example, the subset of the lexicon 5.9 shown in 5.13 contains the paradigms of *dance* and *laugh*. Each of them has been identified as a batch (#3 and #14, respectively). While their string components are different, the feature bundles in these two batches are identical: `=d +k t`, `=d perf`, `=d prog`, `=d v`. This pattern spans across both intransitive verbs in the lexicon.

dance :: `=d v`
danced :: `=d +k t`
danceing :: `=d prog`
dancen :: `=d perf`
dances :: `=d +k t`
laugh :: `=d v`
laughd :: `=d +k t`
laughing :: `=d prog`
laughn :: `=d perf`
laughs :: `=d +k t`

Figure 5.13: A subset of 5.9

To capture this intuition, we introduce the concept of *metabatches*. A metabatch is a pair whose first component is a set of syntactic feature bundles, and whose second component is a set of batches formed from LIs with these feature bundles. Starting with the batches from Table 5.3, the set of (non-singleton) metabatches is as follows:

#	Metabatch	Shared syntax	Shared phonology
1	*dance*, *danced*, *danceing*, *dancen*, *dances*; *laugh*, *laughd*, *laughing*, *laughn*, *laughs*	`=d`	*dance-*; *laugh-*
2	*danced*, *dances*; *laughd*, *laughs*	`=d +k`	*dance-*; *laugh-*
3	*praise*, *praised*, *praiseing*, *praisen*, *praises*; *hug*, *hugd*, *huging*, *hugn*, *hugs*	`=d *k =d`	*praise-*; *hug-*
4	*praised*, *praises*; *hugd*, *hugs*	`=d *k =d +k`	*praise-*; *hug-*

Table 5.4: Metabatches obtained from 5.11 and 5.12

To implement this, we modify the batch formation algorithm as shown in Algorithm 5.7. Just like its predecessor, the new function `Make-Metabatches-Left` utilizes pairwise inter-

section to form batches. However, sets of batches are organized into metabatches according to their syntactic feature bundles.

Function `Make-Metabatches-Left(`Lex`)`
  Data: Lexicon Lex
  Result: Set of metabatches M
  Initialize $M = \varnothing$
  B_{syn} = `Prefix-Syn(`Lex`)`
  B_{phon} = `Prefix-Phon(`Lex`)`
  foreach $b_i = \{s_1 :: \gamma_1, ..., s_n :: \gamma_n\}$ in B_{syn} **do**
    Initialize $B = \varnothing$
    foreach b_j in B_{phon} **do**
      Add $b_i \cap b_j$ to B
    Add $\langle\{\gamma_1, ..., \gamma_n\}, B\rangle$ to M
  return M

Algorithm 5.7: Metabatch formation

Algorithm 5.8 takes care of metabatch decomposition proper. All batches within a metabatch are processed in parallel, producing a single new lexicon but creating a separate fresh category feature and shared morpheme for each batch. The syntactic split point is the same for all batches, whereas the split on the phonological side is decided separately for each. That said, metabatch decomposition steps are greedier, riskier options. In case a metabatch turns out to be a bad step, but some batches within it are good, the algorithm retains ordinary batches and makes separate decomposition steps for each of them. This way, metabatch decomposition steps serve as additional, high-risk and high-reward candidates added to the overall pool.

Function `Decompose-Metabatches-Left`(C, K, Lex)

Data: Set of candidates C, set of known grammars K, lexicon Lex

Result: Updated C and K

$M =$ `Make-Metabatches-Left`(Lex)

foreach $m = \langle\{\gamma_1, ..., \gamma_n\}, B\rangle$ in M **do**

  foreach l_{syn} in `Syn-Range`$(\{\gamma_1, ...\gamma_n\})$ **do**

    $Lex_m = Lex$ `// initialize metabatch candidate`

    foreach $b = \{s_1 :: \delta_1, ..., s_n :: \delta_n\}$ in B **do**

      Let l_{phon} be the length of the longest common prefix of $\{s_1, ...s_n\}$

      $Lex_b = \textbf{ldec}(Lex, b, l_{phon}, l_{syn})$ `// single batch decomposition`

      Add Lex_b to C

      Add Lex_b to K

      $Lex_m = \textbf{ldec}(Lex_m, b, l_{phon}, l_{syn})$ `// metabatch decomposition`

    Add Lex_m to C

    Add Lex_m to K

return C, K

Algorithm 5.8: Left decomposition over metabatches

This chapter has provided a high-level overview of an optimization algorithm for minimalist grammars based on lexical item decomposition and auxiliary operations, followed by a more in-depth discussion of several specific issues. The following chapter reports the results of applying the algorithm to a number of test grammars, as well as the effects of tweaking the method of cost calculation and other parameters.

Chapter 6

Experiments

6.1 Corpus cost and constraints on contraction

As the first line of testing, we use the lexicon introduced in Section 5.3, repeated below. It generates 64 sentences (16 configurations of auxiliaries × 4 lexical verb paradigms), so it is practical to use the entire language as the dataset D.

Mary :: `d -k`	*dance* :: `=d v`	*hug* :: `=d *k =d v`
willd :: `=v +k t`	*danced* :: `=d +k t`	*hugd* :: `=d *k =d +k t`
wills :: `=v +k t`	*dancing* :: `=d prog`	*huging* :: `=d *k =d prog`
have :: `=perf v`	*dancen* :: `=d perf`	*hugn* :: `=d *k =d perf`
haved :: `=perf +k t`	*dances* :: `=d +k t`	*hugs* :: `=d *k =d +k t`
haves :: `=perf +k t`	*laugh* :: `=d v`	*praise* :: `=d *k =d v`
be :: `=prog v`	*laughd* :: `=d +k t`	*praised* :: `=d *k =d +k t`
bed :: `=prog +k t`	*laughing* :: `=d prog`	*praiseing* :: `=d *k =d prog`
ben :: `=prog perf`	*laughn* :: `=d perf`	*praisen* :: `=d *k =d perf`
bes :: `=prog +k t`	*laughs* :: `=d +k t`	*praises* :: `=d *k =d +k t`

Figure 6.1: English auxiliaries and lexical verbs (=5.9)

We start by automatically generating the head-complement graph (for visualization purposes) and calculating the grammar and corpus cost of the input. For the sake of being

explicit, the graph in 6.2 and the rest of head-complement graphs throughout this chapter represent lexical items with no selector features as edges from unlabeled nodes to category features; and edges themselves are labeled with both string components and feature bundles of relevant LIs.

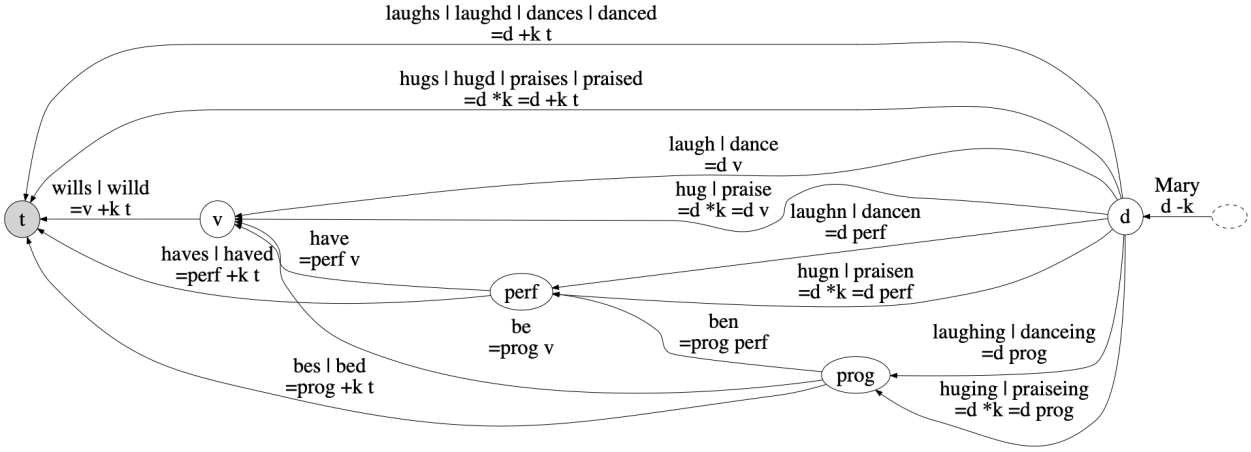
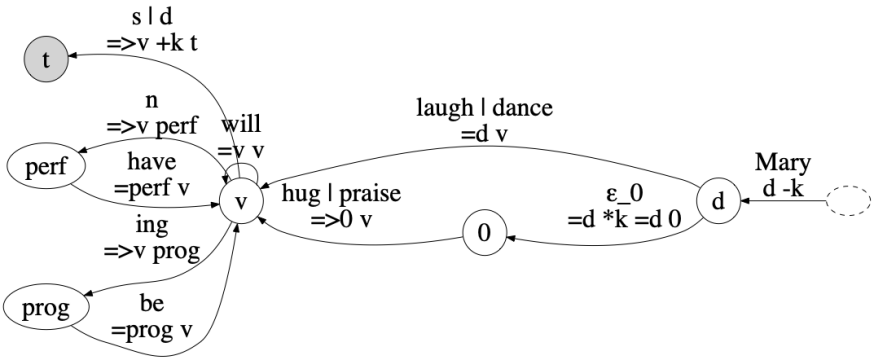
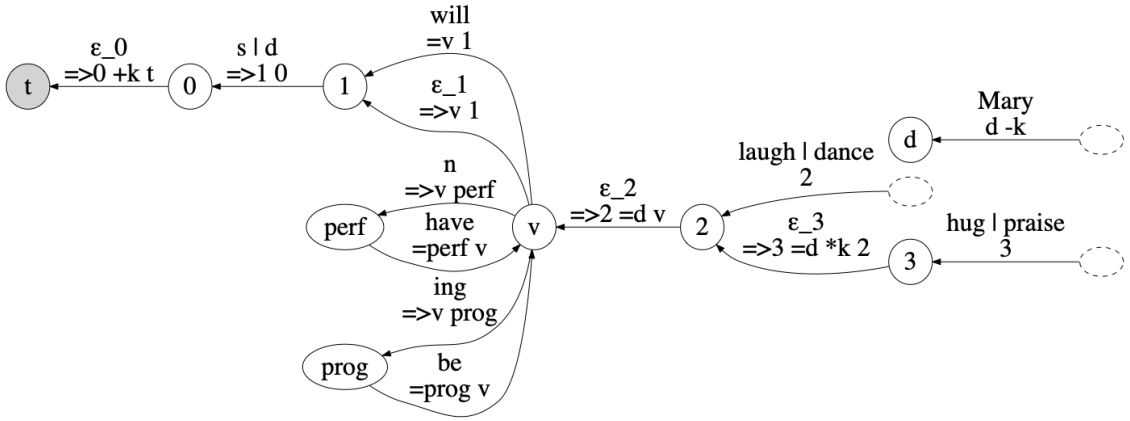


Figure 6.2: Graph of 6.1 before optimization. Grammar: 2001.04 bits, corpus: 410.05 bits

Running the optimization procedure on this input, as described in Chapter 5, yields 6.3a at $bs = 100$ and 6.3b at $bs = 10$. In both cases, the algorithm does achieve significant compression compared to the original in terms of grammar cost: 600.05 and 649.67 bits vs. 2001.04 bits. However, the corpus costs of 499.51 and 464.31 bits are higher than the initial cost of 410.05 bits. This result is reminiscent of the overgenerating grammar from Section 3.2. Some crucial distinctions present in the original have been collapsed into a single category feature, and the hierarchical relationship between `v`, `perf`, and `prog` has been altered. In addition to the original corpus, both new grammars have cycles which lead to the generation of ungrammatical sentences such as **Mary be-s* $(\textit{be-ing})^+$ *laugh-ing* and **Mary have-s* $(\textit{have-n})^+$ *laugh-n*. In addition, 6.3a has another cycle involving the LI *will* :: `=v v`, which allows it to generate **Mary will-s* $(\textit{will})^+$ *laugh*.



(a) $bs = 100$, grammar: 600.05 bits, corpus: 499.51 bits



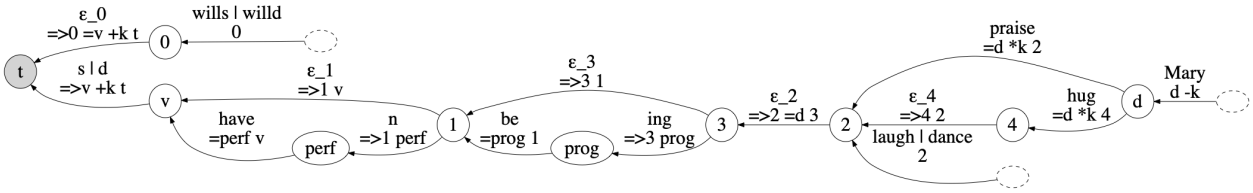
(b) $bs = 10$, grammar: 649.67 bits, corpus: 464.31 bits

Figure 6.3: Optimizing the sum of corpus and grammar

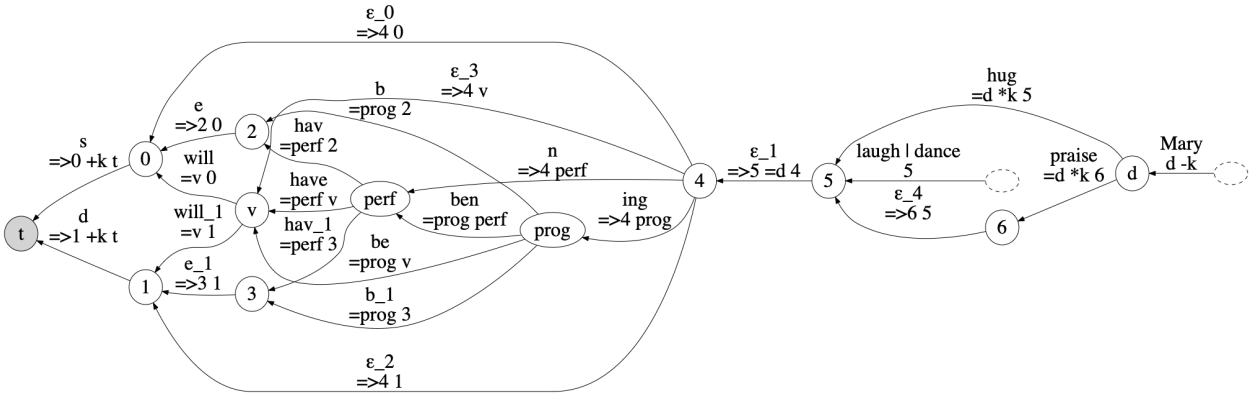
Why did this happen? With respect to the grammar cost, collapsing as many distinctions as possible is indeed the optimal strategy. As we have seen before with both CFGs and MGs, it is the corpus cost that pushes back against overgeneration. However, our dataset – the entire language generated by the original grammar – is finite and relatively small. Because of this, the increase of the corpus cost associated with collapsing too many category distinctions is insufficient to offset the massive reduction to the grammar cost. The result is effectively decided by the grammar cost alone – an unsurprising consequence of using a very small grammar generating a finite language.

If we were to approximate a real-life dataset, the opposite should be the case. A grammar faithfully representing the syntax of a natural language would generate an infinite set of sentences. Since the algorithm would have to work with only a finite subset of this language,

this could be mimicked either by supplying a sufficiently large corpus to balance out the grammar cost, or by calculating the overall cost as an ordered pair of the corpus and grammar cost. In the latter case, the corpus cost is explicitly the primary target of optimization, while the grammar cost serves to break the ties. Applying this strategy to the same input with varying beam sizes produces the grammars in 6.4.



(a) $bs = 100$, grammar: 774.35 bits, corpus: 384.00 bits



(b) $bs = 10$, grammar: 1038.71 bits, corpus: 400.16 bits

Figure 6.4: Optimizing the corpus and grammar as an ordered pair

In general, 6.4a is a reasonable result. This grammar generates the same corpus as the original. Both the grammar and the corpus are compressed compared to the input. The auxiliaries and lexical verbs have been decomposed, and the redundancies mostly eliminated. The somewhat counter-intuitive presence of ϵ :: `=>4 2` is an artifact of the corpus encoding scheme.[19] That said, *wills* and *willd* were not decomposed in the same way as in the previous experiment, leading to incomplete unification of the Tense suffixes *-s* and *-d*. This was a

19. More specifically, the final CFG contains four rules whose left-hand side is (2), one per lexical verb. Because each of the four lexical verbs is used in 16 sentences, the total cost of using these rules is $64 \log_2 4 = 128$ bits. If -ϵ :: `=>4 2` was eliminated by edge contraction, the initial choice would have only three options and cost $\log_2 3 \approx 1.58$ bits per use. However, *hug* and *praise* would end up with identical feature bundles, and their terminal rules would share the left-hand side, incurring an additional 1 bit per sentence involving

suboptimal choice, as it did not decrease the corpus size and prevented the grammar size from being reduced. The under-generalization is even more prominent at $bs = 10$ (6.4b). In both cases, the procedure ended up stopping at a local minimum.

What we want, ideally, is a compromise between the two modes of optimization. The learner should focus on optimizing the grammar cost, like 6.3a, while retaining important feature distinctions and avoiding overgeneration, like 6.4a. To address this problem, let us take a closer look at what causes overgeneration and which properties of the lexicon are indicative of it. We can visualize the entire set of morphological words produced by a lexicon in a way partially similar to a head-complement graph. In a *word graph*, the set of vertices contains all categories in *Base* as well as the designated start and end vertices v_{start} and v_{end}, and each path from v_{start} to v_{end} corresponds to a morphological word.

Definition 6.1: Word graphs and words

Let *Lex* be a minimalist grammar. The *word graph* of *Lex* is a directed multigraph $\langle V, E\rangle$, where $V = Base \cup \{v_{start}, v_{end}\}$, $v_{start}, v_{end} \notin Base$; $\langle \texttt{t}, v_{end}\rangle \in E$; and for each $s :: \alpha\texttt{x}\beta \in Lex$, $\texttt{x} \in Base$:

- If the first feature in α is `=>y`, for some $\texttt{y} \in Base$, then $\langle \texttt{y}, \texttt{x}\rangle \in E$ is an edge labeled with $s :: \alpha\texttt{x}\beta$. Otherwise, $\langle v_{start}, \texttt{x}\rangle \in E$ is an edge labeled with $s :: \alpha\texttt{x}\beta$;
- For each `z` such that $\texttt{z} \in Base$ and `=z` or `z=` is in α, $\langle \texttt{z}, v_{end}\rangle \in E$.

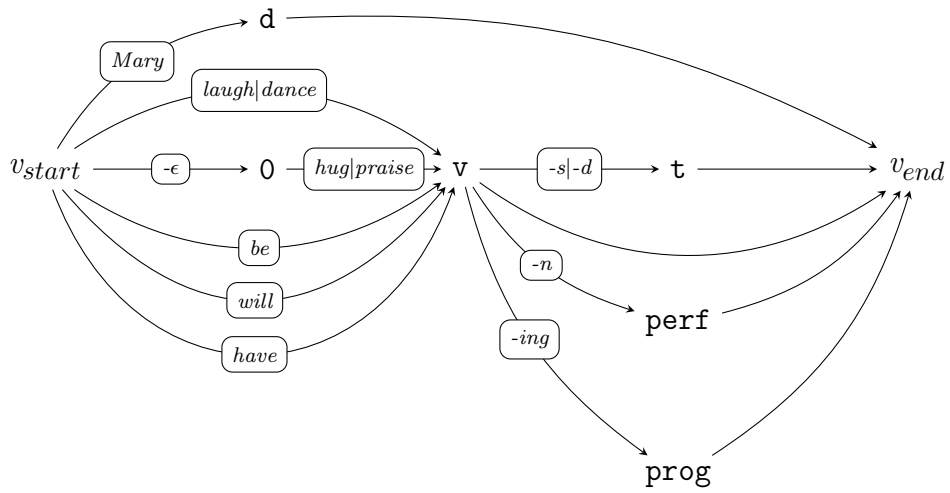
A *word* in *Lex* is the tuple of all edge labels on some path from v_{start} to v_{end} in the word graph of *Lex*.

---

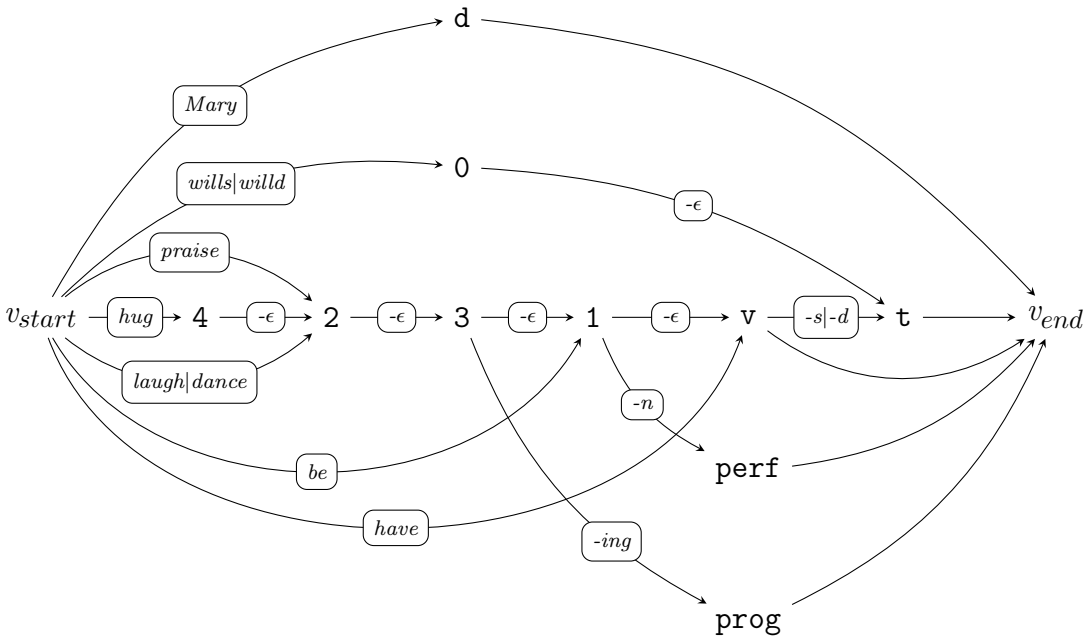
either of these verbs. The cost of these rules across the corpus would be $64 \log_2 3 + 32 \approx 133.44$ bits, which is higher than that of 6.4a.

This could be amended with the following tweak to corpus encoding. For each nonterminal symbol α in the CFG, we would replace the set of all terminal rules $\alpha \to s_1, ..., \alpha \to s_n$ with an intermediate rule $\alpha \to \alpha'$ and new terminal rules $\alpha' \to s_1, ..., \alpha' \to s_n$. The intermediate rule would be annotated with the sum of occurrences of $\alpha \to s_1, ..., \alpha \to s_n$, while the new terminal rules would retain their original usage data. This modification would negate the difference in corpus costs caused by `-ϵ :: =>4 2`, leaving the algorithm free to eliminate it and reduce the grammar cost.

In an input grammar over unsegmented words, the set of morphological words is (trivially) the set of string components of its lexical items. Compare the words in the original lexicon (5.9) to the word graphs of 6.3a and 6.4a.



(a) Optimizing the sum of corpus and grammar (6.3a)



(b) Optimizing the corpus and grammar as an ordered pair (6.4a)

Figure 6.5: Word graphs

The overgenerating lexicon in 6.5a produces a number of morphological words not present in the original grammar: *have-n*, *have-ing*, *be-ing*, *will-n*, and *will-ing*. On the other hand, the set of words in 6.5b is identical to the original.

Recall that overgeneration is caused by unconstrained application of edge contraction. With the above observation is mind, we can mimic the effect of optimizing for corpus size by adding the following heuristic:

- At the preprocessing step, record the set of words in the original lexicon;
- Whenever the MG is modified, update each of the original words to reflect its new path through the word graph;
- Edge contraction is allowed just in case the words in the new lexicon are sufficiently similar to the original words.

The intuitive notion of sufficient similarity can be formalized in a number of ways. A straightforward option pursued here is to require each new word to have identical pronunciation and exhibit similar syntactic behavior[20] to some original word.

With this heuristic in place, it is possible to optimize lexica without consulting the corpus cost at all. Even with a small beam size ($bs = 10$), this configuration leads to the grammar in 6.6. This is the best result so far, with no overgeneration and all redundancies in lexical and auxiliary verbs properly eliminated.

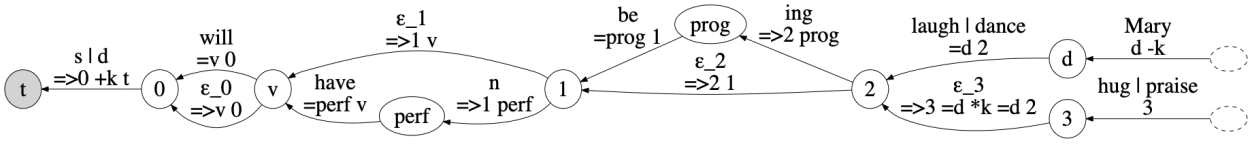


Figure 6.6: Optimizing only the grammar with constraints on edge contraction, $bs = 10$. Grammar: 682.43 bits, corpus: 389.44 bits

20. Roughly, the new word may consist of different morphemes and take a different path through the graph as long as it selects dependents of the same categories in the same order, and the highest morpheme in the word carries the same category as some original word.

In the previous experiments (6.4), overgenerating grammars are added to the candidate pool and eventually ruled out, as they cause an increase in the corpus cost that never pays off. In the setup exemplified by 6.6, the constraint on edge contraction prevents overgenerating grammars from being considered in the first place. This means more promising candidates in the pool, making more efficient use of the limited beam size.

All experiments reported throughout this section are summarized in Table 6.1, with the lowest value of each metric highlighted in **bold**.

	Optimization target	*bs*	Grammar	Corpus	MDL
6.2	Original	–	2001.04	410.05	2411.09
6.3a	Corpus + grammar	100	**600.05**	499.51	1099.56
6.3b	Corpus + grammar	10	649.67	464.31	1113.98
6.4a	⟨Corpus, grammar⟩	100	774.35	**384.00**	1158.35
6.4b	⟨Corpus, grammar⟩	10	1038.71	400.16	1438.87
6.6	Grammar with constraints	10	682.43	389.44	**1071.87**

Table 6.1: Comparison of cost calculating methods (bits)

To recap, the MDL value (sum of corpus and grammar costs) is sensitive to relative sizes of the grammar and corpus; picking it as the optimization target is prone to overgeneration if the corpus is very small. Optimizing for the corpus size first and the grammar second yields better results and resists overgeneration, but the procedure is likely to halt at a local minimum, requiring a larger beam size. The best result was achieved by optimizing for the grammar size in conjunction with a heuristic constraining the application of edge contraction. Note that in this case the MDL value is the lowest; the corpus cost is close to the minimum observed in 6.4a,[21] even though it is not part of the optimization target. This configuration is used in the rest of experiments throughout this chapter.

21. The slightly larger corpus size of 389.44 bits, compared to the minimum of all experiments (384 bits), is caused by a quirk of the encoding scheme; see Footnote 19.

6.2 Passives, complement clauses, raising, and *it*

For the next experiment, we consider a larger fragment of the English grammar. The input MG (6.7) generates a wider range of constructions:

- Multiple proper nouns, common nouns, and determiners;
- The auxiliary system, as in Section 6.1;
- Lexical verbs with different argument structures: intransitive, transitive, and selecting clausal complements (e.g. *Mary declares that John is jumping*);
- Passive constructions with transitive verbs;
- Raising (*Mary seems to smile*) and expletive *it* (*it seems that Mary smiles*).

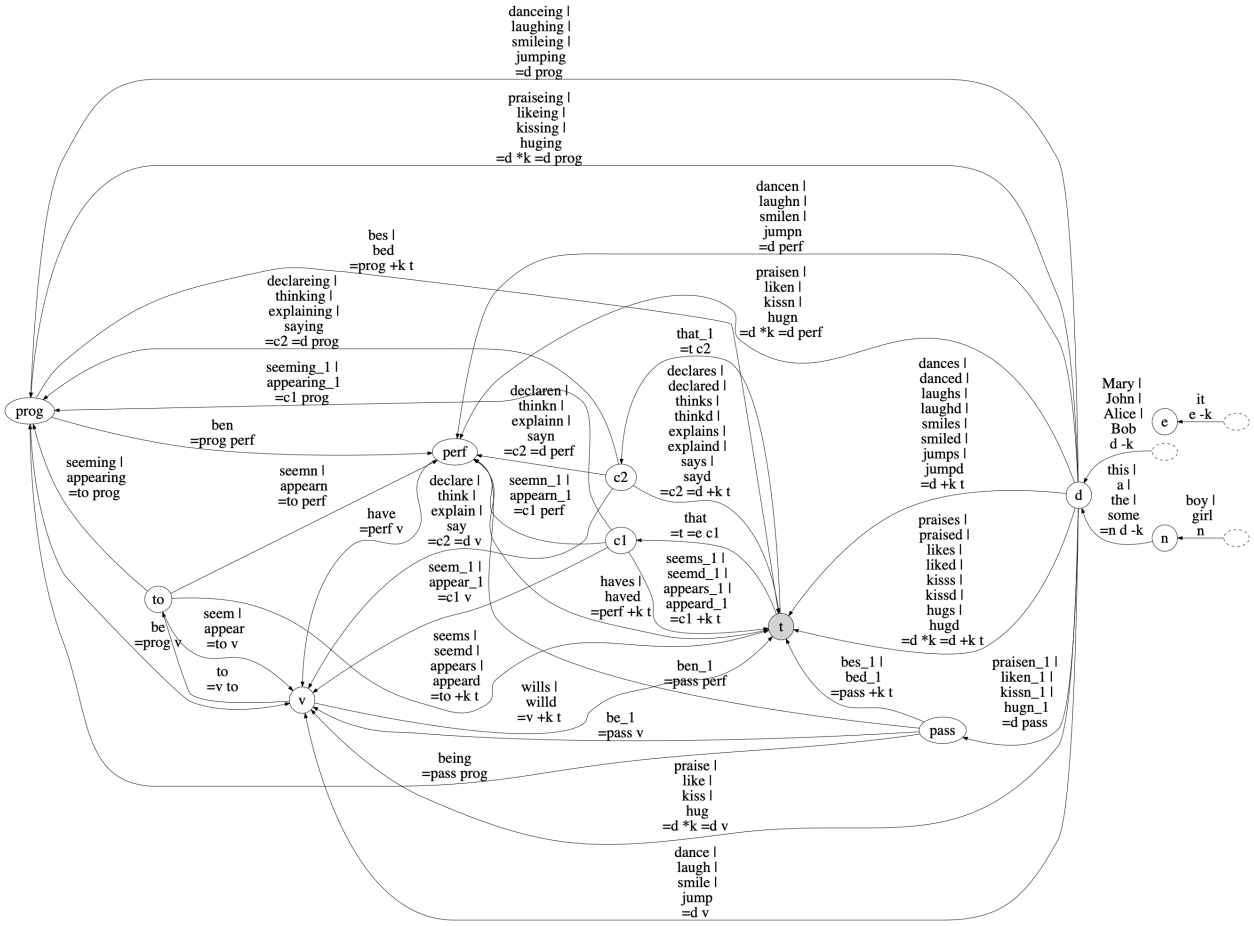


Figure 6.7: Extended fragment before optimization. Grammar: 7572.80 bits

Alice :: `d -k`
Bob :: `d -k`
John :: `d -k`
Mary :: `d -k`
a :: `=n d -k`
some :: `=n d -k`
the :: `=n d -k`
this :: `=n d -k`
boy :: `n`
girl :: `n`

(a) Nouns and determiners

willd :: `=v +k t`
wills :: `=v +k t`
have :: `=perf v`
haved :: `=perf +k t`
haves :: `=perf +k t`
be :: `=prog v`
bed :: `=prog +k t`
ben :: `=prog perf`
bes :: `=prog +k t`

be :: `=pass v`
bed :: `=pass +k t`
being :: `=pass prog`
ben :: `=pass perf`
bes :: `=pass +k t`
it :: `e -k`
that :: `=t =e c1`
that :: `=t c2`
to :: `=v to`

(b) Closed-class LIs

dance :: `=d v`
danced :: `=d +k t`
danceing :: `=d prog`
dancen :: `=d perf`
dances :: `=d +k t`
jump :: ...
laugh :: ...
smile :: ...

(c) Intransitive verbs

declare :: `=c2 v`
declared :: `=c2 +k t`
declareing :: `=c2 prog`
declaren :: `=c2 perf`
declares :: `=c2 +k t`
explain :: ...
think :: ...
say :: ...

(d) Verbs with clausal complements

hug :: `=d *k =d v`
hugd :: `=d *k =d +k t`
huging :: `=d *k =d prog`
hugn :: `=d *k =d perf`
hugn :: `=d pass`
hugs :: `=d *k =d +k t`
kiss :: ...
like :: ...
praise :: ...

(e) Transitive verbs

appear :: `=to v`
appear :: `=to +k t`
appear :: `=to prog`
appear :: `=to perf`
appear :: `=to +k t`
seem :: ...

appear :: `=c1 v`
appear :: `=c1 +k t`
appear :: `=c1 prog`
appear :: `=c1 perf`
appear :: `=c1 +k t`
seem :: ...

(f) Raising verbs

Figure 6.8: Lexical items of 6.7

The LIs of this grammar are listed in 6.10, with some syntactically identical paradigms omitted for space reasons. As before, the input MG is over unsegmented words. Each verb starts out as a full paradigm of unrelated lexical items. In addition, if there are multiple homophones with different syntactic properties, each has to start out as a separate LI. In particular, there are two separate paradigms of *be* (passive and progressive), two paradigms of each raising verb (accounting for raising and expletive *it*), and two copies of *that* (enabling complement clauses and structures with *it*).

As expected, the procedure achieves a significant compression of the grammar. The output grammar is almost four times smaller than the original: 1964.68 bits vs. 7572.80 bits at $bs = 100$. The output graph and lexical items are given in 6.9 and 6.10 respectively; derived trees in 6.11–6.14 provide a number of side-by-side concrete examples before and after optimization.

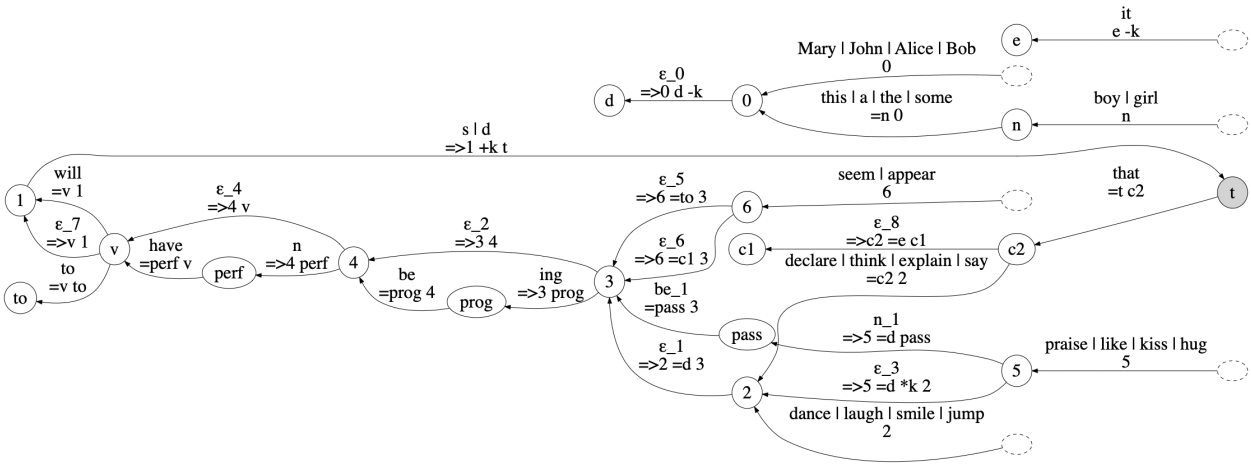


Figure 6.9: Extended fragment after optimization, $bs = 100$. Grammar: 1964.68 bits

The optimized auxiliary system is consistent with previous experiments: all paradigms are decomposed, and hierarchical relations between categories are enforced by category changers. Each lexical verb (6.10c–6.10f) has collapsed into a single LI carrying one of four syntactic feature bundles according to its distribution: `2` for intransitives, `5` for transitives, `6` for raising verbs, and `=c2 2` for verbs with clausal complements. Argument-structure differences between verbs of different categories are encoded as empty LIs. In the nominal domain

(6.10a), the original feature sequence associated with DPs (`d -k`) has been split off into an empty LI, with proper nouns and determiners left with a new feature, `0`. Similarly, most homophonous items have been unified, and their syntactic differences are handled by empty heads. This includes raising verbs in various constructions and the two varieties of *that*. The only homophones (apart from the ones with an empty string component) are the pairs of *be* :: `=pass 3` vs. *be* :: *prog* 4 and *-n* :: `=>4 perf` vs. *-n* :: `=>5 =d pass`.

Alice :: `0`
Bob :: `0`
John :: `0`
Mary :: `0`
a :: `=n 0`
some :: `=n 0`
the :: `=n 0`
this :: `=n 0`
-ϵ :: `=>0 d -k`
boy :: `n`
girl :: `n`

(a) Nouns and determiners

-d :: `=>1 +k t`
-s :: `=>1 +k t`
will :: `=v 1`
-ϵ :: `=>v 1`
have :: `=perf v`
-n :: `=>4 perf`
-ϵ :: `=>4 v`
be :: `=prog 4`
-ing :: `=>3 prog`
-ϵ :: `=>3 4`

be :: `=pass 3`
-n :: `=>5 =d pass`
it :: `e -k`
that :: `=t c2`
-ϵ :: `=>c2 =e c1`
-ϵ :: `=>6 =c1 3`
to :: `=v to`
-ϵ :: `=>6 =to 3`
-ϵ :: `=>2 =d 3`
-ϵ :: `=>5 =d *k 2`

(b) Closed-class LIs

dance :: `2`
jump :: `2`
laugh :: `2`
smile :: `2`

(c) Intransitive verbs

declare :: `=c2 2`
explain :: `=c2 2`
think :: `=c2 2`
say :: `=c2 2`

(d) Verbs with clausal complements

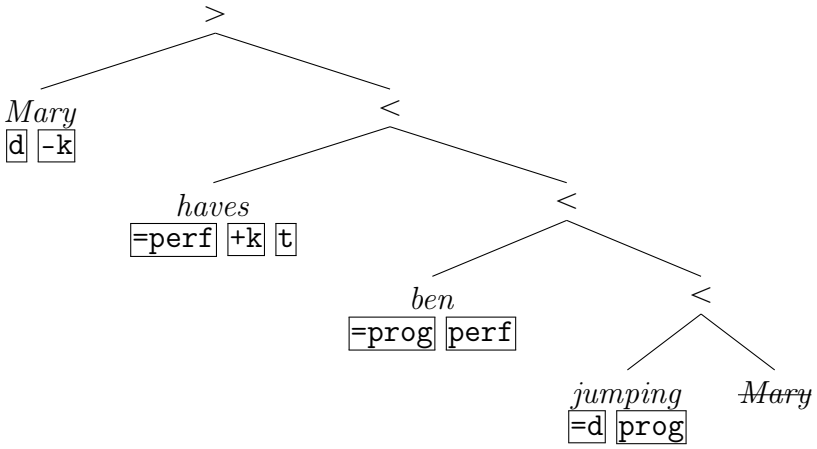
hug :: `5`
kiss :: `5`
like :: `5`
praise :: `5`

(e) Transitive verbs

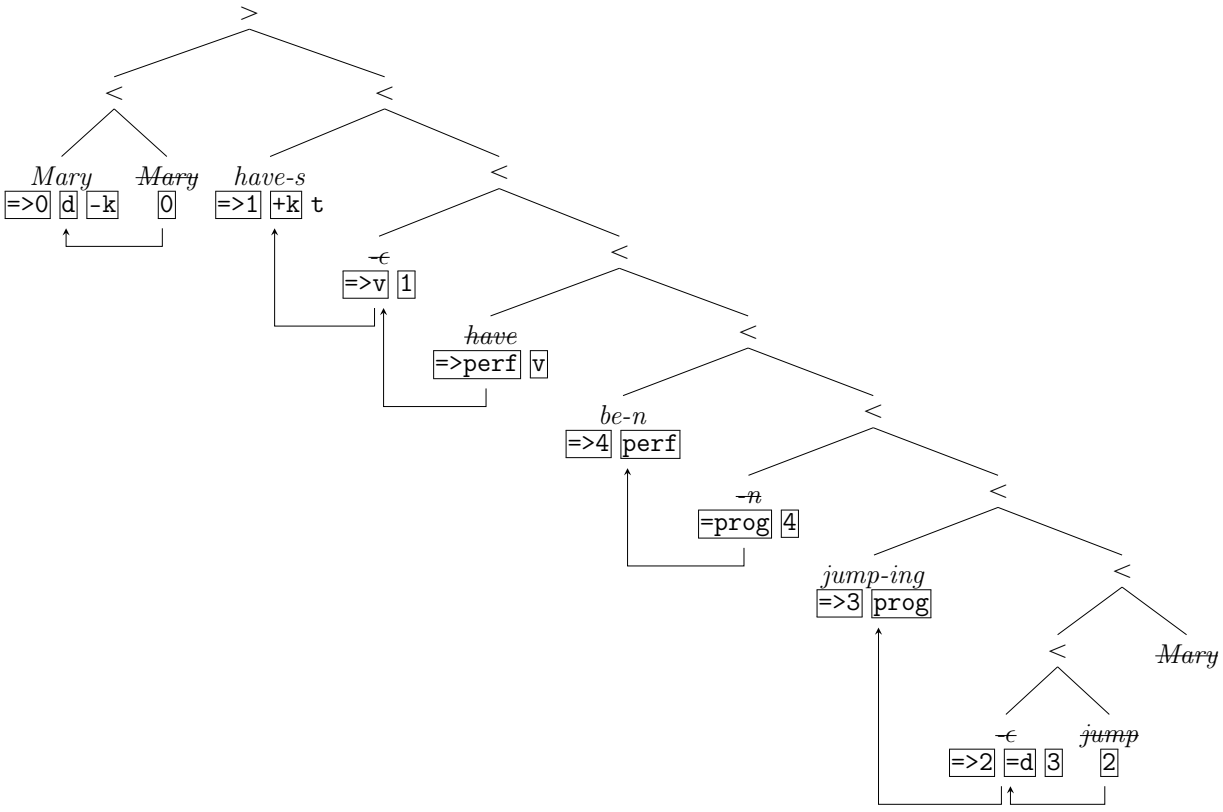
appear :: `6`
seem :: `6`

(f) Raising verbs

Figure 6.10: Lexical items of 6.9

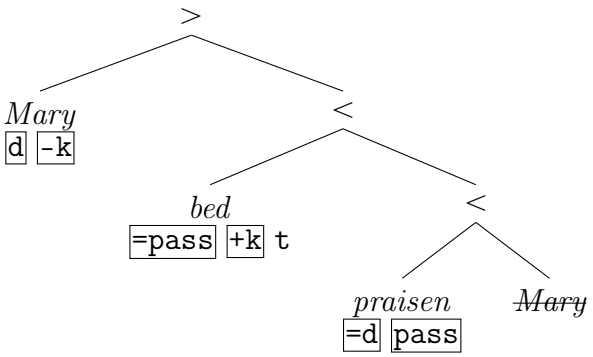


(a) *Mary haves ben jumping* (using 6.7)

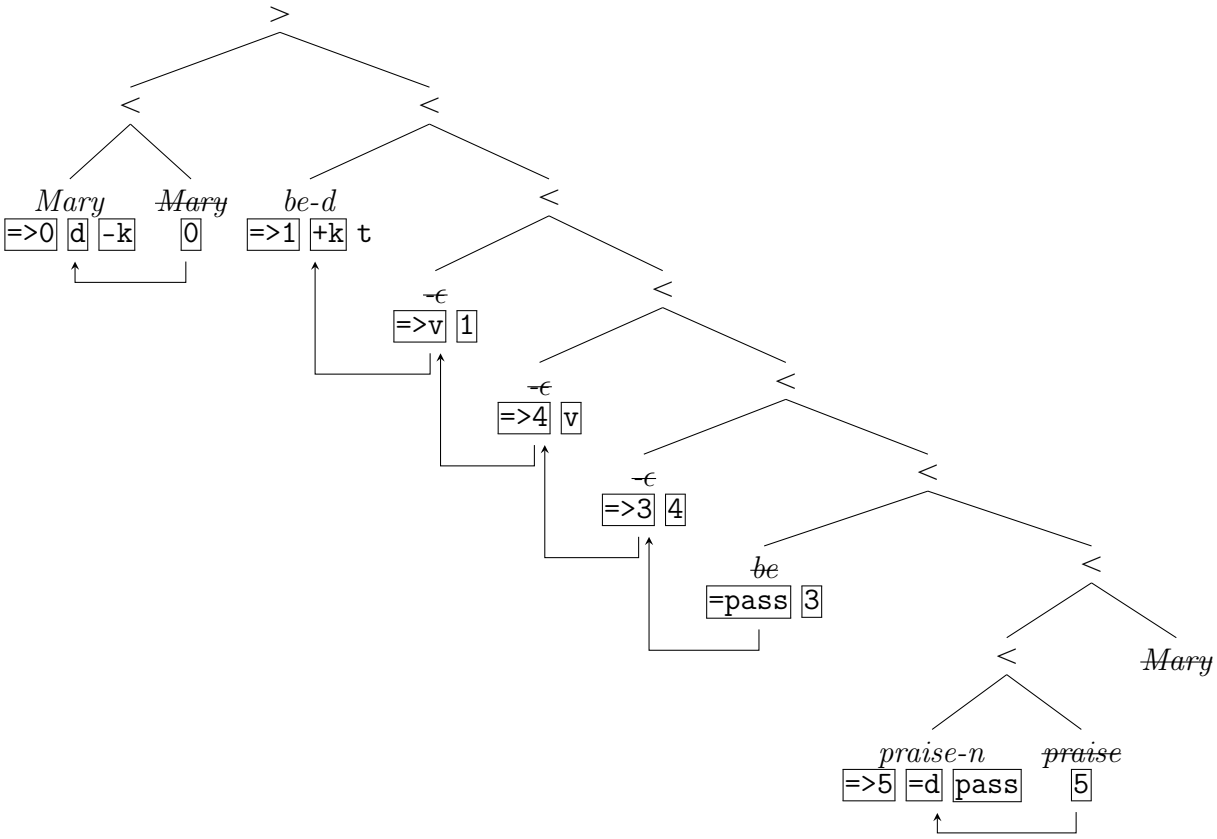


(b) *Mary have-s be-n jump-ing* (using 6.9)

Figure 6.11: Before and after: auxiliaries

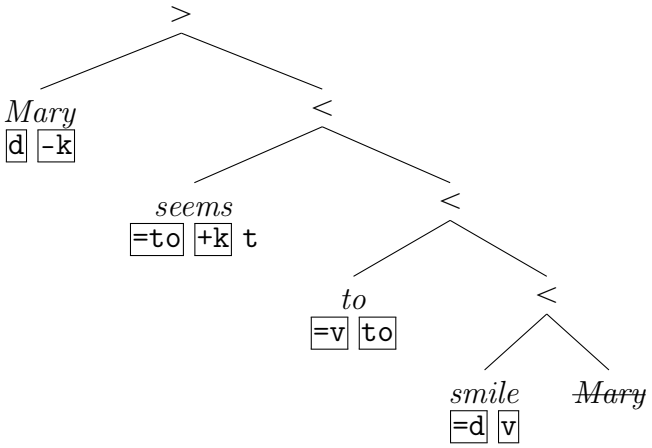


(a) *Mary bed praisen* (using 6.7)

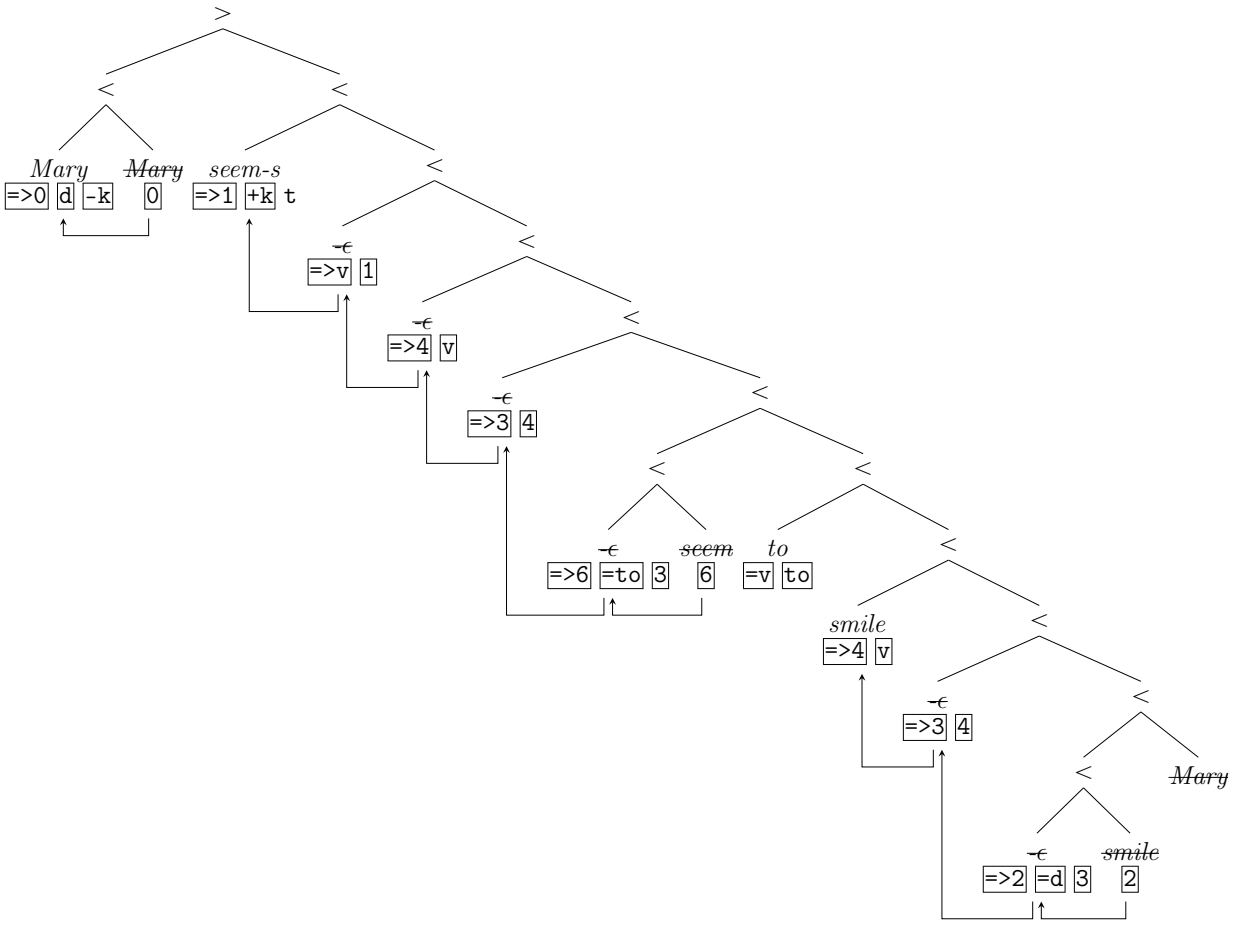


(b) *Mary be-d praise-n* (using 6.9)

Figure 6.12: Before and after: passives

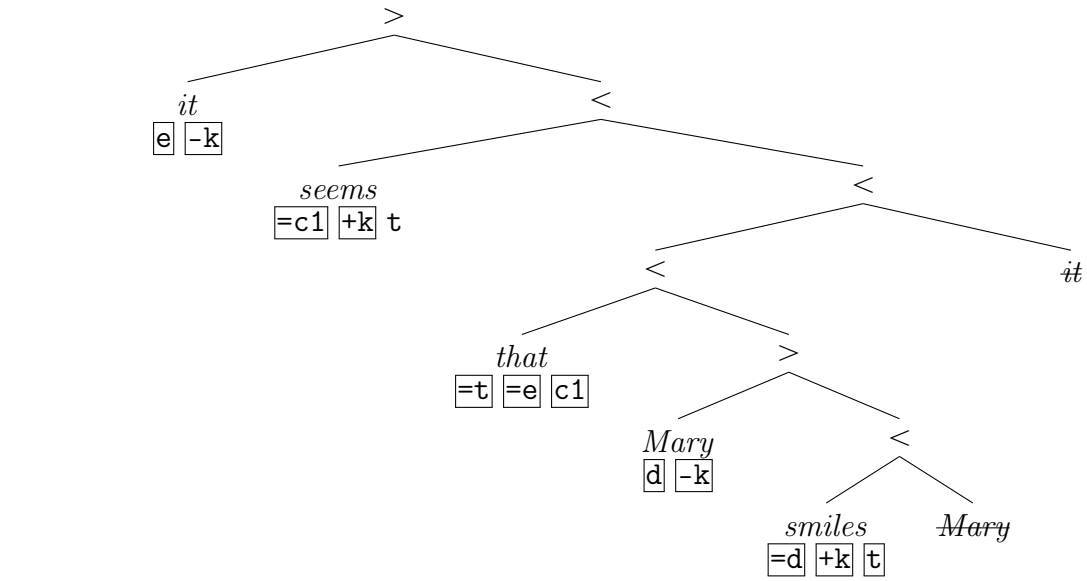


(a) *Mary seems to smile* (using 6.7)

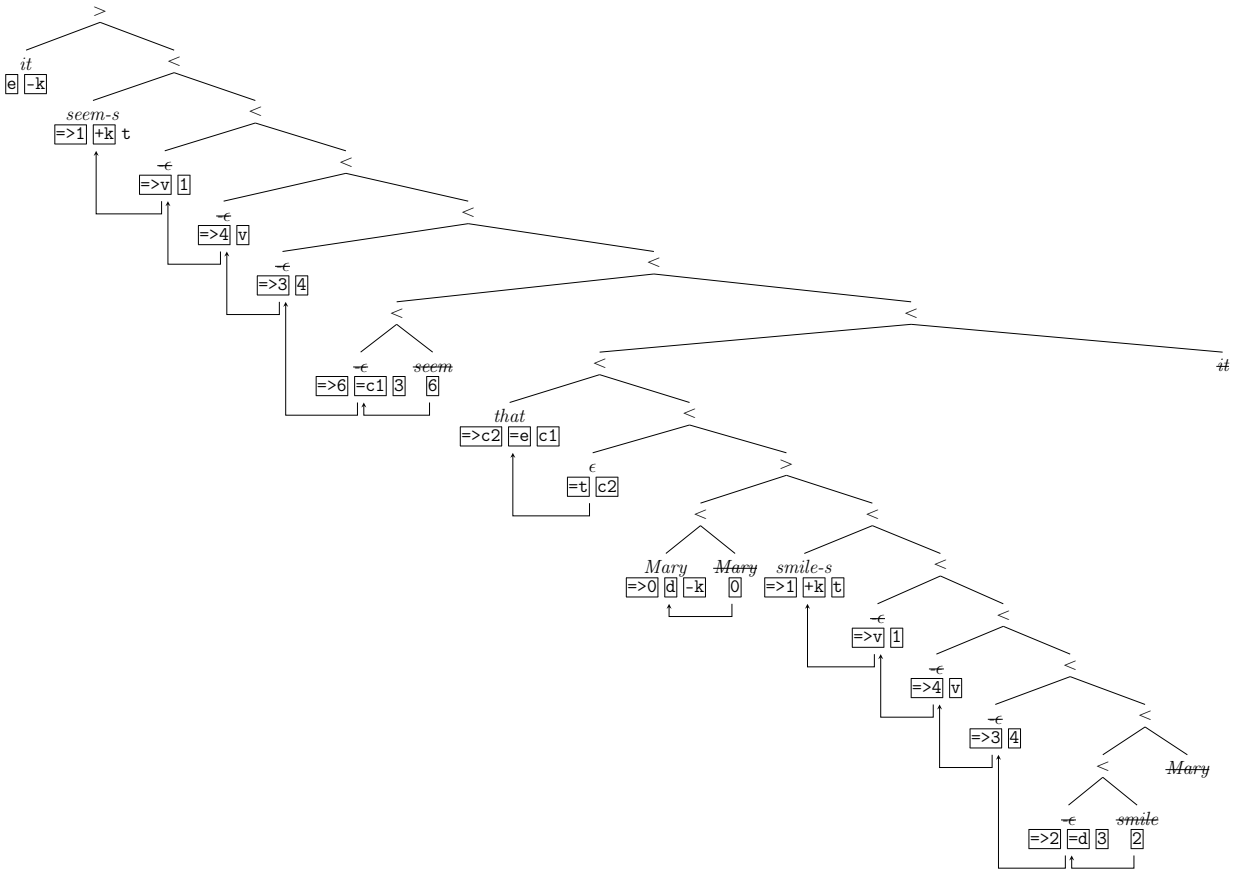


(b) *Mary seem-s to smile* (using 6.9)

Figure 6.13: Before and after: raising



(a) *it seems that Mary smiles* (using 6.7)



(b) *it seem-s that Mary smile-s* (using 6.9)

Figure 6.14: Before and after: expletive *it*

In general, the learner tends to push syntactic differences between original LIs into empty functional heads. In a number of cases this is taken to the logical conclusion: all proper nouns and most lexical verbs carry a bundle of one category feature, and their syntactic requirements are supplied by categorizing heads which they are selected by. This is consistent with the idea of acategorial roots having to merge with a categorizing head carrying the rest of features, which is a general assumption in Distributed Morphology Marantz (1997); Embick and Marantz (2008). From the quantitative perspective, associating a large number of string components with the shortest possible feature bundle is a beneficial strategy. In a larger grammar, we can expect this strategy to be pursued especially aggressively with open-class items, since they would make up the bulk of words with identical syntactic distribution.

Many of the decisions made by the algorithm in this experiment align with a linguist's intuition. That said, the output grammar does present some instances of counter-intuitive lack of decomposition. One example is an inconsistency in how lexical verbs are decomposed: while all others end up reduced to single-feature roots, verbs with clausal complements select their complements directly, each retaining a feature bundle of `=c2 2`. Another is the presence of two suffixes which select transitive verbs and independently supply their internal argument, -ϵ :: `=>5 =d *k 2` and the passive -n :: `=>5 =d 2`.

Nothing in the procedure prohibits the generation of a grammar where both of these apparent issues are resolved. A single **ldec** step would suffice to reduce *declare* :: `=c2 2`, *explain* :: `=c2 2`, *think* :: `=c2 2`, *say* :: `=c2 2` to *declare* :: `x`, *explain* :: `x`, *think* :: `x`, *say* :: `x`, selected by -ϵ :: `=>x =c2 2`. Another would ensure that the internal argument of the verb is merged in by a separate head -ϵ :: `=>5 =d y`, which in turn would be selected by -ϵ :: `=>y *k 2` for active constructions and by -ϵ :: `=>y pass` for passives. However, each of these steps would slightly increase the grammar size (to 1994.57 bits if both are implemented) rather than reduce it. Both problems boil down to a single issue: any decomposition step has to be worth it in the quantitative sense.

6.3 Lexical selection of PPs

In the experiments reported so far, we have seen the optimizing algorithm consistently identify syntactic and/or phonological differences between words and instantiate them as separate (sometimes silent) lexical items, in order to factor out what the words had in common. An interesting test case for this behavior comes from lexical selection (l-selection, Pesetsky 1991) of prepositional phrases.

As reported by Merchant (2019), idiosyncratic selectional properties of most roots in English remain uniform across different realizations of the root. For example, the root $\sqrt{\text{RELI}}$ appears with a PP complement headed by the preposition *on* whether it is realized as the verb *rely* (3a), noun *reliance* (3b), or adjective *reliant* (3c). However, some roots break this pattern. There is a large class of cases, such as $\sqrt{\text{FEAR}}$, where the verb takes a direct object (4a) while the noun and adjective select a PP with *of* (4b, 4c). Furthermore, selectional properties of a number of roots vary across multiple realizations. This is the case with $\sqrt{\text{PRD}}$, which takes an *on*-PP as the verb *pride oneself* (5a), an *in*-PP as the noun *pride* (5b), and an *of*-PP as the adjective *proud* (5c).

(3) a. They rely on oil.

b. Their reliance on oil is well-known.

c. They are reliant on oil.

(4) a. Abby fears dark spaces.

b. Abby's fear of dark spaces is well known.

c. Abby is fearful of dark spaces.

(5) a. She prides herself on/*in/*of her thoroughness.

b. Her pride *on/in/*of her thoroughness is understandable.

c. She is proud *on/*in/of her thoroughness.

(Merchant 2019, pp. 327, 329)

Regular examples like (3) provide an argument for roots being acategorial. They can be captured with an analysis like (6.15a), where the root itself selects for its complement, and the resulting structure is in turn selected by a head categorizing it as a noun, verb, or adjective. This analysis, however, does not work for roots with non-uniform selectional properties such as those in (4) and (5), where the choice of a complement alternates across realizations. To account for these, Merchant proposes a solution illustrated by 6.15b. Under this analysis, categorizing heads come with two pieces of information about selection: which roots they can combine with, and which PPs are compatible with those roots. For instance, the nominalizer N_{in} first selects for a compatible root (such as $\sqrt{\text{PRD}}$) and then for an *in*-PP.

(a) Root selects for its complement PP

(b) Nominalizer selects for the root and PP

Figure 6.15: Two possible structures for l-selection of PPs (adapted from Merchant 2019)

Root	V	N	A (-ful)
boast	of, about	of, about	of, about
disdain respect	DO	for	of
doubt	(DO)	of, about	of, about
fear neglect scorn	(DO)	of	of
hope	for	for	for

Table 6.2: V-N-A tuples and argument types

Will a similar analysis emerge in the process of grammar optimization? In order to focus on l-selection, let us consider a small grammar fragment featuring (mostly) concatenative morphology. Table 6.2 presents a sample of complete ⟨verb, noun, adjective⟩ triples selected from examples and database information presented in (Merchant 2019). The verbs and nouns don't carry any overt categorizing morphology, whereas the adjectives are formed with the suffix *-ful*. The sample includes roots with uniform complements (*boast*, *hope*); members of the large class alternating between direct objects for verbs and *of*-PPs for everything else (*doubt*, *fear*, *neglect*, *scorn*); and other non-uniform roots (*disdain*, *respect*, *doubt*). An MG over unsegmented words including realizations of these roots is given below, with all lexical items listed in 6.16.

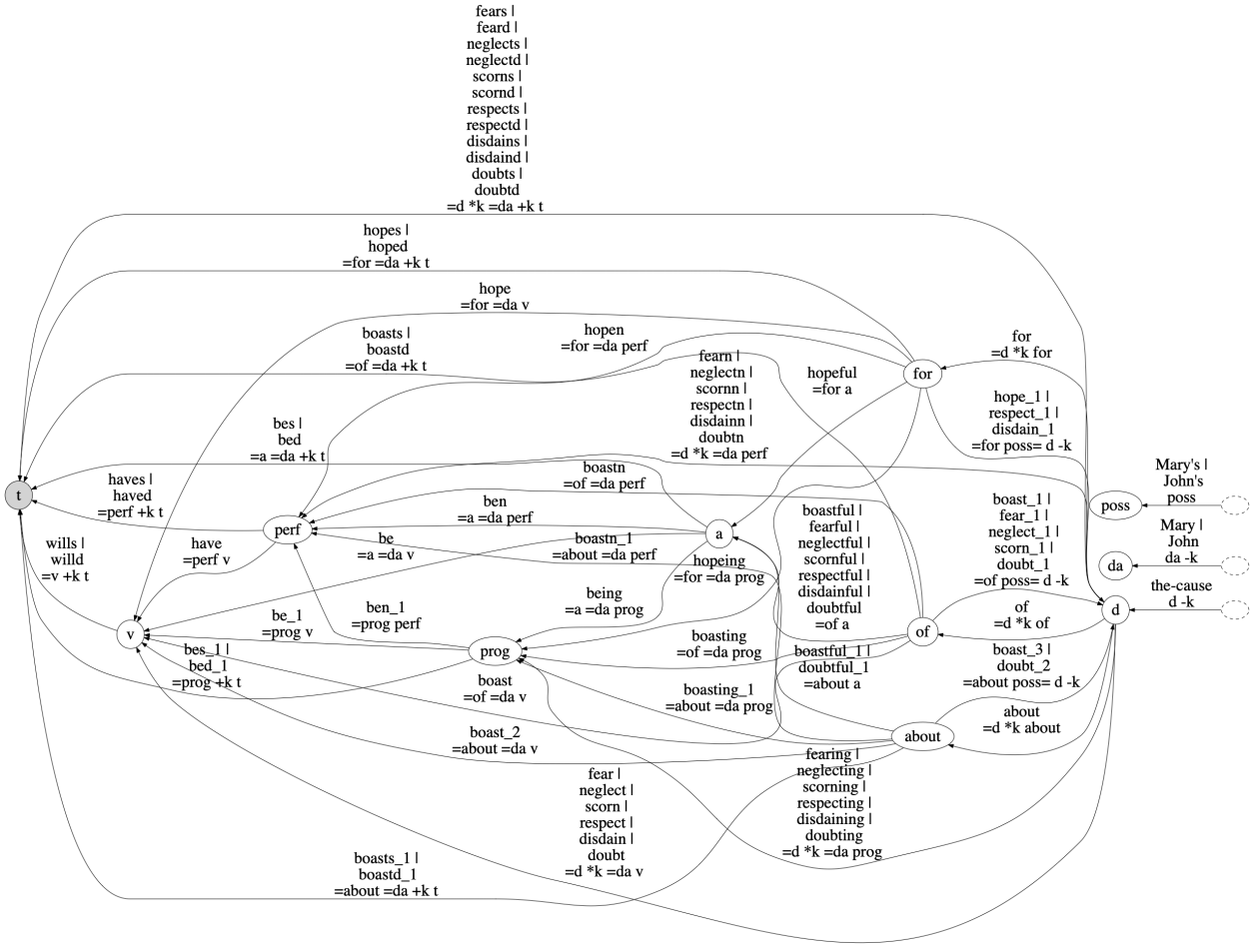


Figure 6.16: l-selection before optimization. Grammar: 6716.65 bits

John :: `da -k`
John's :: `poss`
Mary :: `da -k`
Mary's :: `poss`
the cause :: `d -k`
about :: `=d *k about`
for :: `=d *k for`
of :: `=d *k of`

(a) Nouns and prepositions

willd :: `=v +k t`
wills :: `=v +k t`
have :: `=perf v`
haved :: `=perf +k t`
haves :: `=perf +k t`
be :: `=prog v`
bed :: `=prog +k t`
ben :: `=prog perf`
bes :: `=prog +k t`

be :: `=a =da v`
bed :: `=a =da +k t`
being :: `=a =da prog`
ben :: `=a =da perf`
bes :: `=a =da +k t`

(b) Auxiliaries and copulas

boast :: `=about =da v`
boastd :: `=about =da +k t`
boasting :: `=about =da prog`
boastn :: `=about =da perf`
boasts :: `=about =da +k t`
boast :: `=about poss= d -k`
boastful :: `=about a`

boast :: `=of =da v`
boastd :: `=of =da +k t`
boasting :: `=of =da prog`
boastn :: `=of =da perf`
boasts :: `=of =da +k t`
boast :: `=of poss= d -k`
boastful :: `=of a`

(c) *boast*

disdain :: `=d *k =da v`
disdaind :: `=d *k =da +k t`
disdaining :: `=d *k =da prog`
disdainn :: `=d *k =da perf`
disdains :: `=d *k =da +k t`
disdain :: `=for poss= d -k`
disdainful :: `=of a`
respect :: ...

(d) *disdain*, *respect*

doubt :: `=d *k =da v`
doubtd :: `=d *k =da +k t`
doubting :: `=d *k =da prog`
doubtn :: `=d *k =da perf`
doubts :: `=d *k =da +k t`
doubt :: `=about poss= d -k`
doubtful :: `=about a`
doubt :: `=of poss= d -k`
doubtful :: `=of a`

(e) *doubt*

fear :: `=d *k =da v`
feard :: `=d *k =da +k t`
fearing :: `=d *k =da prog`
fearn :: `=d *k =da perf`
fears :: `=d *k =da +k t`
fear :: `=of poss= d -k`
fearful :: `=of a`
neglect :: ...
scorn :: ...

(f) *fear*, *neglect*, *scorn*

hope :: `=for =da v`
hoped :: `=for =da +k t`
hopeing :: `=for =da prog`
hopen :: `=for =da perf`
hopes :: `=for =da +k t`
hope :: `=for poss= d -k`
hopeful :: `=for a`

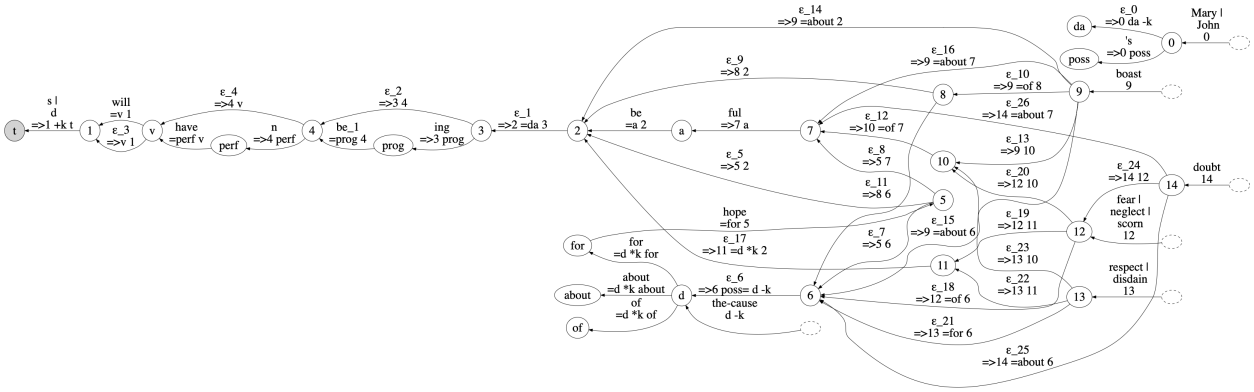
(g) *hope*

Figure 6.17: Lexical items of 6.16

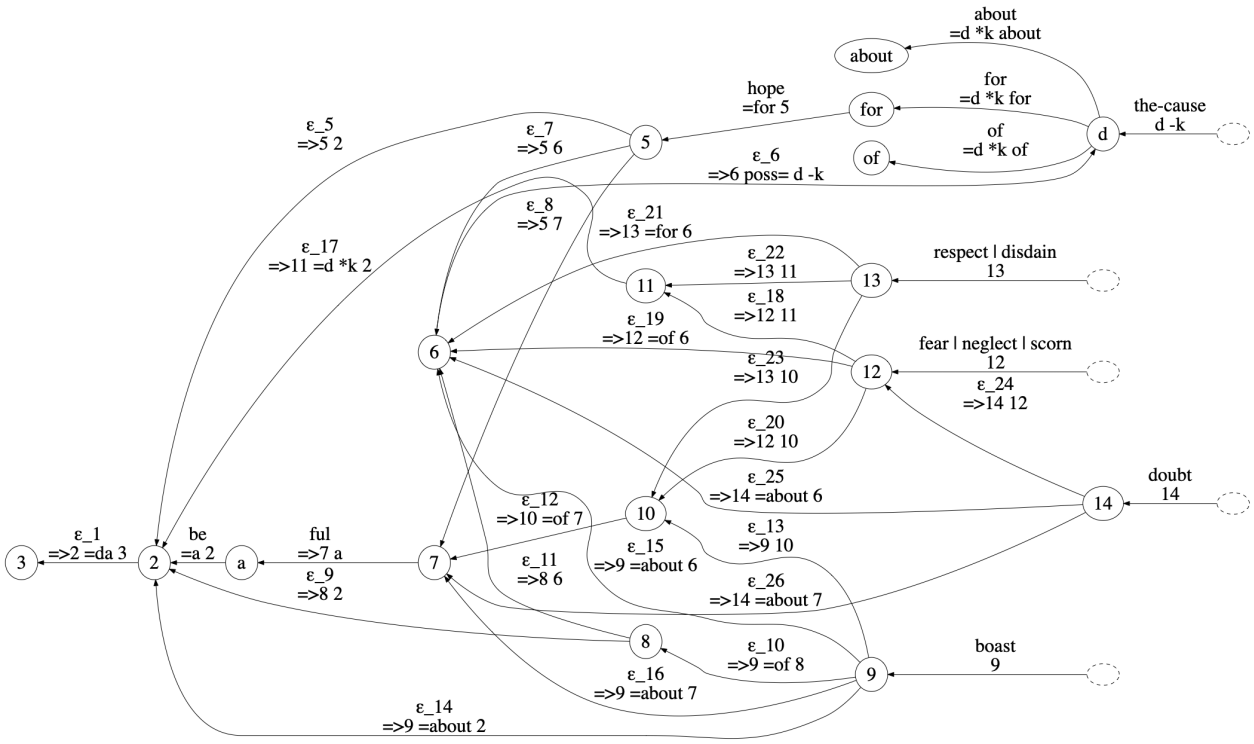
The language generated by this grammar is infinite, since complex DPs can, in turn, be selected by verbs or prepositions. Some sentences from this language are given below:

Mary hopes for the cause;
Mary haves ben respectful of the cause;
Mary bes boasting about John's fear of the cause;
Mary wills be disdainful of the cause;
Mary doubts John's respect for the cause;
Mary bes scornful of John's neglect of Mary's hope for the cause.

Optimization with a large beam size ($bs = 500$) yields the following compressed grammar.



(a) Full graph



(b) Close-up of verbs and categorizers

Figure 6.18: l-selection after optimization, $bs = 500$. Grammar: 2200.30 bits

-d :: `=>1 +k t`
-s :: `=>1 +k t`
will :: `=v 1`
-ϵ :: `=>v 1`

John :: `0`
Mary :: `0`
-'s :: `=>0 poss`
-ϵ :: `=>0 da -k`
the cause :: `d -k`
about :: `=d *k about`
for :: `=d *k for`
of :: `=d *k of`

have :: `=perf v`
-n :: `=>4 perf`
-ϵ :: `=>4 v`
be :: `=prog 4`
-ing :: `=>3 prog`
-ϵ :: `=>3 4`
be :: `=a 2`
-ϵ :: `=>a =da 3`

boast :: `9`
disdain :: `13`
doubt :: `14`
fear :: `12`
hope :: `=for 5`
neglect :: `12`
respect :: `13`
scorn :: `12`

(a) Nouns and prepositions

(b) Auxiliaries and copulas

(c) Lexical verbs

-ful :: `=>7 a`
-ϵ :: `=>5 7`
-ϵ :: `=>9 =about 7`
-ϵ :: `=>10 =of 7`
-ϵ :: `=>14 =about 7`

-ϵ :: `=>2 =da 3`
-ϵ :: `=>11 =d *k 2`
-ϵ :: `=>5 2`
-ϵ :: `=>8 2`
-ϵ :: `=>9 =about 2`

-ϵ :: `=>6 poss= d -k`
-ϵ :: `=>5 6`
-ϵ :: `=>8 6`
-ϵ :: `=>9 =about 6`
-ϵ :: `=>12 =of 6`
-ϵ :: `=>13 =for 6`
-ϵ :: `=>14 =about 6`

-ϵ :: `=>9 =of 8`
-ϵ :: `=>9 10`
-ϵ :: `=>12 11`
-ϵ :: `=>12 10`
-ϵ :: `=>13 11`
-ϵ :: `=>13 10`
-ϵ :: `=>14 12`

(d) Categorizers and category changers

Figure 6.19: Lexical items of 6.18

Apart from expected changes in the nominal and auxiliary domains (6.19a, 6.19b), this grammar features extensive modifications to how roots are categorized. All roots except *hope* have been reduced to a single category (6.19c). Among the remaining LIs, three are easily identifiable as categorizing heads. The verbalizer *-ϵ* :: `=>2 =da 3` is responsible for the verb's external argument, the nominalizer *-ϵ* :: `=>6 poss= d -k` merges in a possessive phrase such as *John's* and assigns the entire structure the typical DP feature sequence of `d -k`, and the adjectivizer *-ful* :: `=>7 a` provides the overt affix and the category `a`.

The rest of LIs in 6.19d fall into two broad classes: category changers and empty heads selecting first one of the new categories, then a PP with a specific preposition. To form a clearer picture of what role each of them plays, we need to consider what these newly created features stand for. A brief summary is provided in Table 6.3.

Category	Interpretation
5	*hope*
9	*boast*
12	*fear*, *neglect*, *scorn*
13	*disdain*, *respect*
14	*doubt*
2	compatible with the verbalizer -ϵ :: `=>2 =da 3`
6	compatible with the nominalizer -ϵ :: `=>6 poss= d -k`
7	compatible with the adjectivizer *-ful* :: `=>7 a`
8	verbal and nominal *boast* with an *of*-PP argument
10	adjectives with an *of* PP argument
11	verbs with a direct object

Table 6.3: Interpretation of features in 6.18

The first major group of features (`5`, `9`, `12`, `13`, `14`) stands for roots. Reducing open-class items to shorter feature bundles is an optimization strategy we have encountered before. Roots with identical selectional properties (*fear*, *neglect*, *scorn*; *disdain*, *respect*) have been assigned the same category. Most of the roots don't select their arguments directly, instead relying on whatever LIs select them to provide the necessary features. The only exception is *hope* :: `=for 5`. This root uniformly selects *for*-PPs across all realizations, and is the only one in the sample to do so – which makes it a suboptimal decision to decompose it further. That said, not all cases of uniform l-selection are treated in the same way. The other such root (*boast* :: `9`) carries a single category, and its selectional properties are handled by other

LIs. One plausible explanation is that the dataset is very small, and *every* root is effectively treated as idiosyncratic. With a larger grammar, we might be able to see a more uniform treatment of frequently occurring patterns.

The second group (`2`, `6`, `7`) is responsible for categorization. Any expression of category `2`, whatever LI it is headed by, is compatible with (i.e. can be selected by) the verbalizing head. This includes verbs which take a direct object, as well as those taking a PP argument, such as *boast* and *hope*. Similarly, `6` represents nouns, and `7` adjectives. Finally, the third group (`8`, `10`, `11`) picks up on more minor patterns, which are nevertheless quantitatively worthy of a separate category.

With this in mind, any valid configuration of a root, category, and argument type can be assembled with a specific combination of empty LIs introducing PPs and category changers, summarized in Table 6.4. Note that there are multiple heads selecting PPs, rather than only one per preposition type. Under Merchant's (2019) analysis, each categorizing head is annotated with its argument type and a list of compatible roots. Minimalist grammars encode this type of choice by having a separate lexical item for each option.[22]

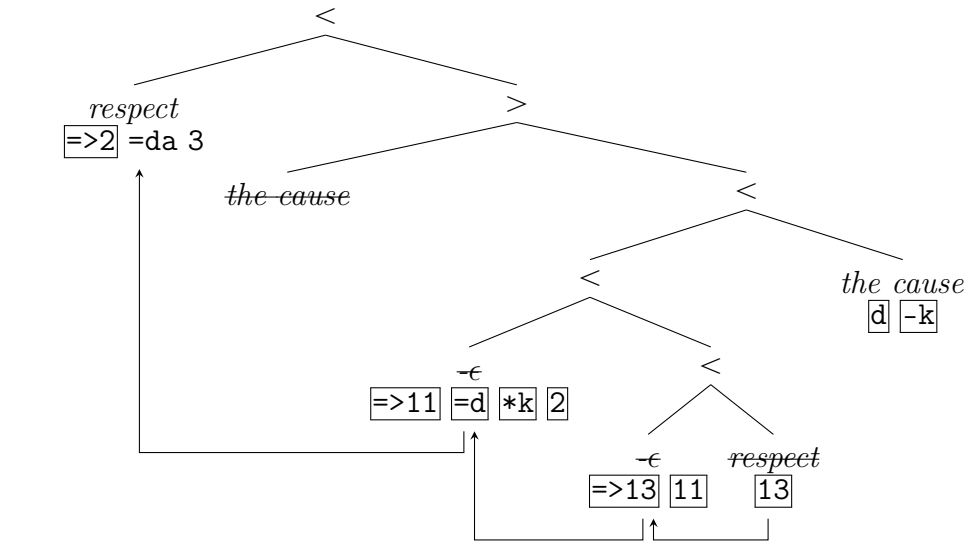
Similarly, category changers are the MGs' way of encoding distributional similarities and differences between roots. For example, *doubt* can take dependents of the same type as *fear*/*neglect*/*scorn*, and additionally can select an *about*-PP when realized as a noun or adjective – unlike *fear*-type roots. This difference warrants a separate category `14` for *doubt*. However, the presence of -ϵ `::` `=>14 12` allows *doubt* to piggy-back on the system in place for *fear*-type verbs (of category `12`) for the shared part of its selectional options. Essentially, this category changer encodes the statement that *doubt* can be found in all contexts of *fear*, as well as some of its own.

For a concrete illustration, trees in 6.20 demonstrate how the optimized grammar derives the three realizations of *respect* and their non-uniform selectional properties.

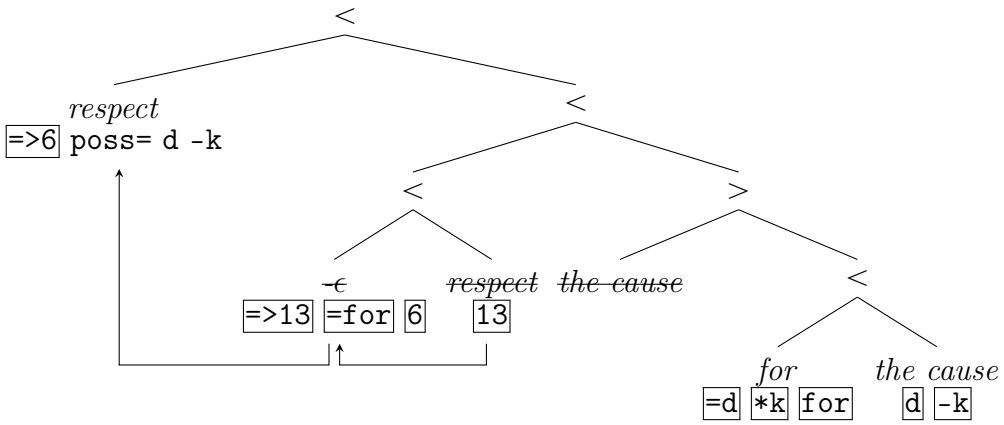
22. A good question to ask is whether we could leverage the idea of a list to inform our encoding scheme. For example, it may be possible to encourage multiple LIs which only differ in one syntactic feature by recording everything they share only once, making them cheaper to encode.

Root	Argument	Category	Implementation in LIs
boast	about	V	*boast* :: 9, -ϵ :: =>9 =about 2
		N	*boast* :: 9, -ϵ :: =>9 =about 6
		A	*boast* :: 9, -ϵ :: =>9 =about 7
	of	V	*boast* :: 9, -ϵ :: =>9 =of 8, -ϵ :: =>8 2
		N	*boast* :: 9, -ϵ :: =>9 =of 8, -ϵ :: =>8 6
		A	*boast* :: 9, -ϵ :: =>9 10, -ϵ :: =>10 =of 7
disdain	DO	V	*disdain* :: 13, -ϵ :: =>13 11, -ϵ :: =>11 =d *k 2
	for	N	*disdain* :: 13, -ϵ :: =>13 =for 6
	of	A	*disdain* :: 13, -ϵ :: =>13 10, -ϵ :: =>10 =of 7
respect	DO	V	*respect* :: 13, -ϵ :: =>13 11, -ϵ :: =>11 =d *k 2
	for	N	*respect* :: 13, -ϵ :: =>13 =for 6
	of	A	*respect* :: 13, -ϵ :: =>13 10, -ϵ :: =>10 =of 7
doubt	DO	V	*doubt* :: 14, -ϵ :: =>14 12, -ϵ :: =>12 11, -ϵ :: =>11 =d *k 2
	about	N	*doubt* :: 14, -ϵ :: =>14 =about 6
		A	*doubt* :: 14, -ϵ :: =>14 =about 7
	of	N	*doubt* :: 14, -ϵ :: =>14 12, -ϵ :: =>12 =of 6
		A	*doubt* :: 14, -ϵ :: =>14 12, -ϵ :: =>12 10, -ϵ :: =>10 =of 7
fear	DO	V	*fear* :: 12, -ϵ :: =>12 11, -ϵ :: =>11 =d *k 2
	of	N	*fear* :: 12, -ϵ :: =>12 =of 6
	of	A	*fear* :: 12, -ϵ :: =>12 10, -ϵ :: =>10 =of 7
neglect	DO	V	*neglect* :: 12, -ϵ :: =>12 11, -ϵ :: =>11 =d *k 2
	of	N	*neglect* :: 12, -ϵ :: =>12 =of 6
	of	A	*neglect* :: 12, -ϵ :: =>12 10, -ϵ :: =>10 =of 7
scorn	DO	V	*scorn* :: 12, -ϵ :: =>12 11, -ϵ :: =>11 =d *k 2
	of	N	*scorn* :: 12, -ϵ :: =>12 =of 6
	of	A	*scorn* :: 12, -ϵ :: =>12 10, -ϵ :: =>10 =of 7
hope	for	V	*hope* :: =for 5, -ϵ :: =>5 2
	for	N	*hope* :: =for 5, -ϵ :: =>5 6
	for	A	*hope* :: =for 5, -ϵ :: =>5 7

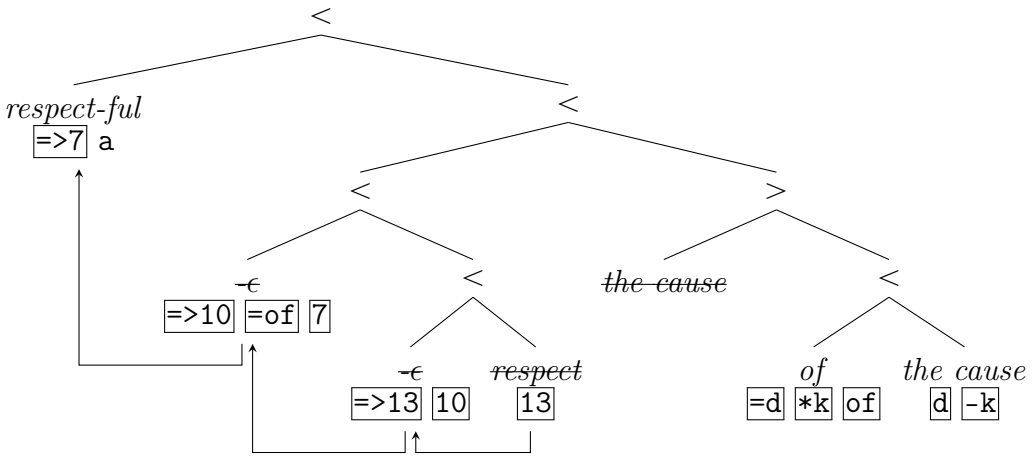
Table 6.4: Verbs and categorizing heads in 6.18



(a) Verb



(b) Noun



(c) Adjective

Figure 6.20: Selectional variability of *respect* (using 6.18)

Some properties of this optimized grammar stem from those of the formalism. Selection in (our chosen version of) MGs is symmetric, which means roots cannot be truly acategorial; as long as heads selecting them are compatible with some roots but not all of them, the roots themselves must carry different category features to enable this distinction. Similarly, each categorizer is represented as a combination of multiple lexical items, where in a theoretical work it might be a single head assigned different surface realizations by additional vocabulary insertion rules. However, modulo these (largely notational) differences, the emerging MG is fundamentally similar to the proposal of (Merchant 2019). The algorithm has arrived at the conclusion that (most) roots don't select, whereas categorizing heads do; and it did so by optimizing a quantitative measure.

6.4 Beyond pseudo-English

Throughout this dissertation, we have been working under the assumptions in Subsection 2.3.2, reducing all morphology to suffixation and simplifying English words to fully concatenative strings. An interesting question to ask at this point is whether the same procedure can find any patterns in natural, un-simplified English.

Table 6.5 shows some (orthographical) alternations in English verbs. It includes three paradigm types: non-alternating (*jump*), with a stem-final alternating *e* (*dance*), and with a stem-final consonant doubling before *-ed* and *-ing* (*brag*). Since these alternations are concatenative, we can expect the optimization algorithm to be capable of picking up these patterns.

Type	Example paradigm			
Non-alternating	jump	jumps	jumped	jumping
Alternating -e	dance	dances	danced	dancing
Alternating -g	brag	brags	bragged	bragging

Table 6.5: Allomorphy in English verbs

The input grammar fragment includes verbal paradigms of the three types shown in Table 6.5, three verbs per type, with both transitive and intransitive verbs represented. For this experiment, the string components of both lexical verbs and auxiliaries are left unmodified. The graph and LI list of the input are given in 6.21 and 6.22, respectively.

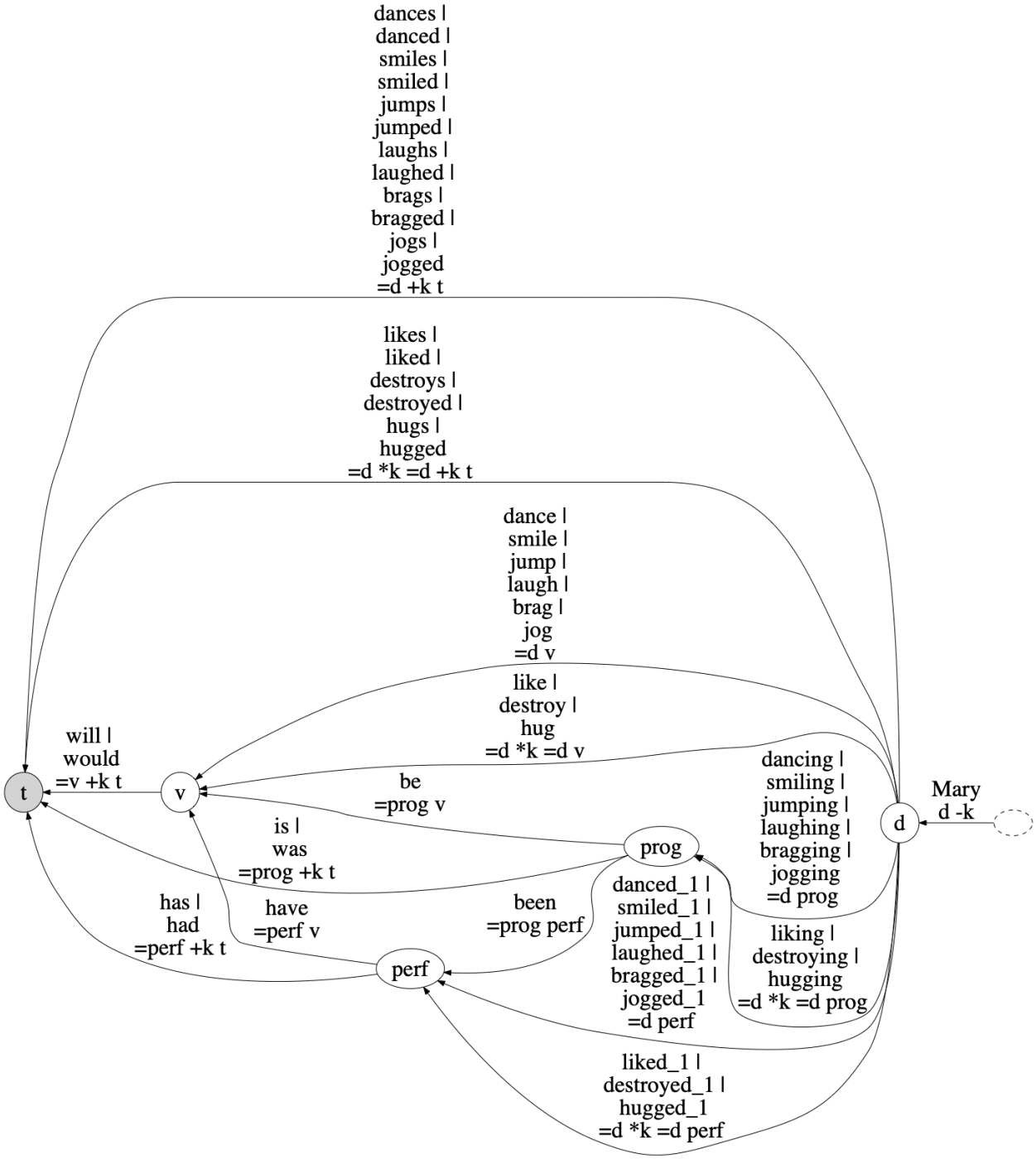


Figure 6.21: Natural English fragment before optimization. Grammar: 3666.81 bits

		be :: `=prog v`
Mary :: `d -k`	*have* :: `=perf v`	*been* :: `=prog perf`
will :: `=v +k t`	*had* :: `=perf +k t`	*is* :: `=prog +k t`
would :: `=v +k t`	*has* :: `=perf +k t`	*was* :: `=prog +k t`

(a) Nouns and auxiliaries

jump :: `=d v`	*dance* :: `=d v`	*brag* :: `=d v`
jumps :: `=d +k t`	*dances* :: `=d +k t`	*brags* :: `=d +k t`
jumped :: `=d +k t`	*danced* :: `=d +k t`	*bragged* :: `=d +k t`
jumped :: `=d perf`	*danced* :: `=d perf`	*bragged* :: `=d perf`
jumping :: `=d prog`	*dancing* :: `=d prog`	*bragging* :: `=d prog`
laugh :: ...	*smile* :: ...	*jog* :: ...

(b) Intransitive verbs

destroy :: `=d *k =d v`	*like* :: `=d *k =d v`	*hug* :: `=d *k =d v`
destroys :: `=d *k =d +k t`	*likes* :: `=d *k =d +k t`	*hugs* :: `=d *k =d +k t`
destroyed :: `=d *k =d +k t`	*liked* :: `=d *k =d +k t`	*hugged* :: `=d *k =d +k t`
destroyed :: `=d *k =d perf`	*liked* :: `=d *k =d perf`	*hugged* :: `=d *k =d perf`
destroying :: `=d *k =d prog`	*liking* :: `=d *k =d prog`	*hugging* :: `=d *k =d prog`

(c) Transitive verbs

Figure 6.22: Lexical items of 6.21

The output of this experiment ($bs = 500$) is shown in 6.23 and 6.24. As before, the grammar has been significantly compressed: from 3666.81 to 1414.03 bits. Auxiliaries (6.24a) have been left mostly intact; unlike pseudo-English, in their natural form they do not have enough in common phonologically to warrant decomposition. Lexical verbs, on the other hand, have been decomposed.[23] For each verbal paradigm, the algorithm has factored out a stem shared by all words in the paradigm (6.24c). Similarly, it has split off the material on the right edge of the verbs, forming a single set of suffixes compatible with all stems (6.24b).

23. This is reminiscent of the distinction drawn by Lasnik (1995) that auxiliary *be* and *have* are pulled from the lexicon fully inflected, whereas English lexical verbs are bare in the lexicon.

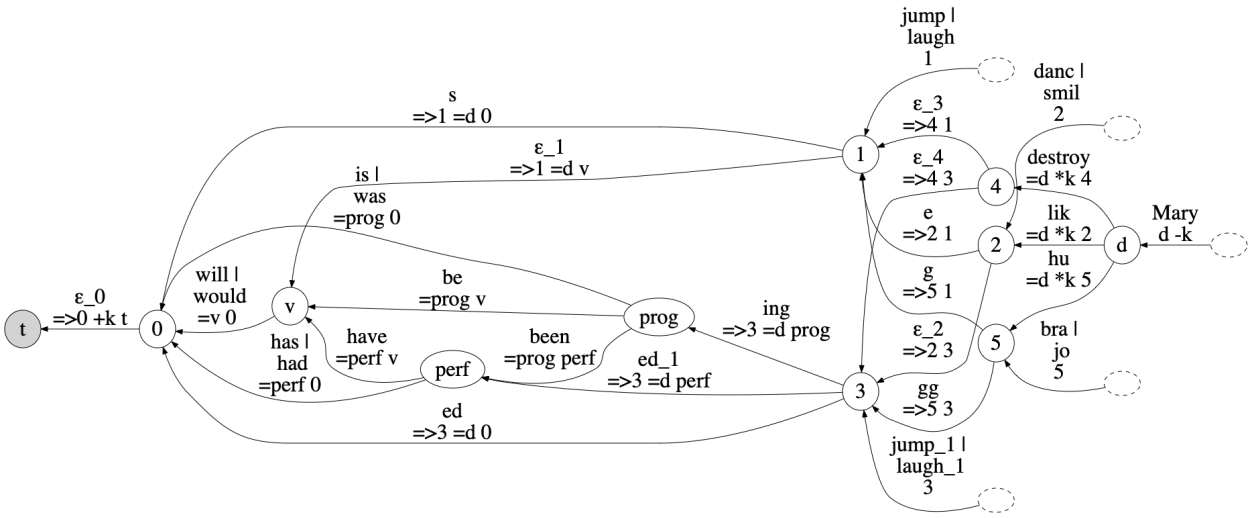


Figure 6.23: Natural English fragment after optimization, $bs = 500$. Grammar: 1414.03 bits

Mary :: `d -k`		
will :: `=v 0`	*be* :: `=prog v`	*-ϵ* :: `=>1 =d v`
would :: `=v 0`	*been* :: `=prog perf`	*-s* :: `=>1 =d 0`
have :: `=perf v`	*is* :: `=prog 0`	*-ed* :: `=>3 =d perf`
had :: `=perf 0`	*was* :: `=prog 0`	*-ed* :: `=>3 =d 0`
has :: `=perf 0`	*-ϵ* :: `=>0 +k t`	*-ing* :: `=>3 =d prog`

(a) Nouns and auxiliaries

(b) Suffixes

jump :: `1`		*-e* :: `=>2 1`
jump :: `3`	*bra* :: `5`	*-ϵ* :: `=>4 1`
laugh :: `1`	*jo* :: `5`	*-g* :: `=>5 1`
laugh :: `3`	*destroy* :: `=d *k 4`	*-ϵ* :: `=>2 3`
danc :: `2`	*lik* :: `=d *k 2`	*-ϵ* :: `=>4 3`
smil :: `2`	*hu* :: `=d *k 5`	*-gg* :: `=>5 3`

(c) Verbs

(d) Allomorphy

Figure 6.24: Lexical items of 6.23

Just as in Section 6.3, each of the newly formed categories has a distinct interpretation, summarized in Table 6.6. The categories `2`, `4`, `5` encode three classes of verb stems grouped by alternation pattern. Similarly, there are two classes of suffixes (`1` and `3`), which differ in

what stems they are compatible with.[24] For each stem type (`2`, `4`, `5`) there are two lexical items (6.24d) which select the stem and are in turn selected by the suffixes, taking care of allomorphy.

Category	Interpretation
`1`	compatible with *-ϵ*, *-s*
`3`	compatible with *-ed*, *-ing*
`2`	stems followed by *-ϵ*, *-s*, *-ed*, *-ing*
`4`	stems followed by *-e*, *-es*, *-ed*, *-ing*
`5`	stems followed by *-g*, *-gs*, *-gged*, *-gging*

Table 6.6: Interpretation of features in 6.23

Note that two verbs, *jump* and *laugh*, have not been reduced to `4`, even though they belong to the same type as *destroy* :: `=d *k 4`. Instead, each appears as two copies bearing the categories `1` and `3` that can be directly selected by the suffixes. This outcome exposes a limitation of the batch formation algorithm, but also characterizes the two verbs as the default case. These verbs are intransitive, and their paradigms are made distinct from others by their *lack* of alternations. There is nothing they have in common, syntactically or morphologically, that would enable *jump* and *laugh* to form a batch with *destroy* to the exclusion of other verbs.

This experiment serves to show that the optimization algorithm is not limited to the somewhat artificial pseudo-English data discussed earlier. When presented with natural, un-simplified words, it did leave the highly suppletive auxiliaries intact. However, the concatenative elements of the input grammar – namely, lexical verbs – underwent decomposition, resulting in a compressed description of allomorphy patterns.

24. The two-way distinction holds, because *-ϵ* and *-s* behave identically with respect to the nine verbs in the lexicon, as do *-ed* and *-ing*. If the lexicon contained the paradigm of *kiss* – which exemplifies yet another alternation pattern – we would expect the algorithm to draw an additional distinction between *-s* and *-ϵ*: in the case of *kiss*, the alternating *-e* is present with the former, but not the latter.

Chapter 7

Conclusions

Summary of results

In this dissertation I have investigated the possibility of comparing, evaluating, and improving syntactic analyses on quantitative grounds. The results reported here are threefold.

- Even within the same framework, such as Chomsky's (1995, 2000) Minimalist Program, there is enough room for alternative accounts of the same observed language data. I have shown how specific proposals stated as minimalist grammars (Stabler 1997) can be compared with the help of an evaluation measure inspired by Minimum Description Length (Rissanen 1978), and how different predictions made by these proposals translate into quantifiable differences;

- Lexical item decomposition (Kobele 2018, to appear) offers a formal characterization of linguistic generalizations, defined as phonological and/or syntactic commonalities across multiple lexical items. Building upon this idea, I have defined a toolkit of three basic operations over minimalist grammars, designed to identify redundancies in a naive minimalist grammar over words, instantiating them as new lexical items. With these operations feeding each other, I have demonstrated how concrete, familiar

generalizations can be obtained through repeated application of easily interpretable steps;

- Finally, I have implemented an optimization algorithm with the three operations at its core. I have used this implementation on a fragment of English including the auxiliary system, verbs with different argument structures, and raising. The resulting grammar was both compressed with respect to the evaluation measure, and in alignment with a linguist's intuition. I have followed this up with experiments focusing on l-selection of prepositional phrases and on verb stem alternations in natural English. In all cases, the algorithm has shown a strong tendency to assign single-category feature bundles to roots and to push phonological and/or syntactic differences into newly derived functional heads.

MG optimization and previous work

The optimization procedure is a mostly self-contained module, ready to be plugged into a larger system or combined with others to build a pipeline for processing linguistic data. This is illustrated by 7.1. The scheme highlights two avenues for possible interaction with other research in the field of grammatical inference: morphology and input generation.

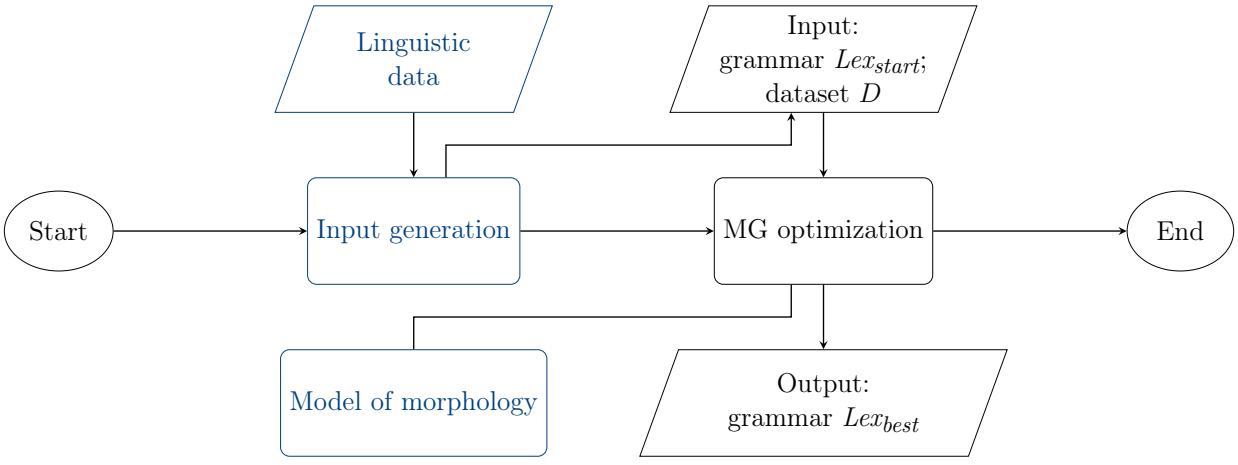


Figure 7.1: The pipeline of MG optimization

First, MG optimization starts with a grammar over unsegmented words. A natural next step would be to pair it with a learning algorithm capable of producing such an input. Two existing lines of research on acquiring knowledge of syntax (represented as a minimalist lexicon) from psychologically plausible data exemplify the potential for synergy.

In a proposal by Kobele et al. (2002) and Stabler et al. (2003), the learner is given ordered and directed dependencies, as well as words segmented into morphemes. The learner's task is to assign a (fresh) label to each relation, forming lexical items, and determine which feature distinctions should be kept and which need to be unified. The pressure for unification comes from a restriction on the number of homophonous lexical items (Kanazawa 1995). Without the segmentation requirement, this approach would produce a naive MG, still capable of generating the original corpus but agnostic of morphological structure.

More recently, work by Indurkhya (2019, 2020) focuses on a scenario with less supervision. The input consists of sentences annotated with syntactic relations, including predicate-argument relations, semantic roles of arguments, and subject-verb agreement. This information is encoded as constraints on the parse tree. Minimalist grammars inferred in this way correctly establish relations between lexical items via **move** and head movement, postulating empty heads where necessary. This output could be fed as input to MG optimization to target remaining redundancies in the lexicon and discover structure within words.

Second, MG optimization works best when there is enough syntactic information to draw on. With respect to string components, it relies on a very rudimentary model of morphology. All affixes are taken to be suffixes, and most of the experiments reported here use simplified inputs in fully concatenative pseudo-English to let the procedure focus on syntactic distinctions. Section 6.4 shows that decomposition is capable, to an extent, of identifying and compressing allomorphy in natural English; but this is essentially an attempt by syntax to analyze phenomena which belong in the domain of morphology. One direction

of future work would involve plugging in a more sophisticated model of morphology to inform both batch formation and cost calculation.[25]

Word segmentation and morpheme discovery based on corpus data, without syntactic features to draw on, fall under the umbrella of unsupervised learning of morphology. Existing work in this area (see e.g. an overview in Goldsmith et al. 2017) makes for a natural complement to MG optimization. One example is the *Linguistica* algorithm (Goldsmith 2001, 2006). It relies on distributional information and the MDL principle to extract a description of morphology from a corpus in an unsupervised scenario. This description is represented as *signatures*, or pairs consisting of a set of stems and a set of suffixes such that every stem is compatible with every suffix. For example, the full paradigms of *jump* and *laugh* could be encoded as the set of stems {*jump*, *laugh*} linked to the suffixes {-ϵ, -*ed*, -*ing*, -*s*}. Linguistica is capable of learning extensive morphology systems, such as the prefixes and suffixes of Swahili (Goldsmith and Mpiranya to appear). Supplying a description of morphology to MG optimization as part of input would take the pressure off the batch formation module, allowing it to focus on what it is good at: identifying syntactically distinct lexical items.

The future

What are the next steps? In the previous section, I have outlined a few possibilities for combining MG optimization with other work in the same field. Within the optimization algorithm itself, there are two immediate areas with room for improvement:

- **Syntax formalism.** This dissertation uses a relatively simple, bare-bones version of minimalist grammars for the sake of conceptual clarity. A more sophisticated implementation of MGs could be used to bring the results closer to those of theoretical

25. To depart from concatenation, we would need to revert to generalized lexical item decomposition as shown in 4.3. Then each word in the original dataset would correspond to a morphological equation specifying which LIs it contains, and our theory of morphology would supply the operation(s) for building the words from the LIs' string components.

syntax. A natural starting point would be to extend the formalism with some variety of generalized head movement, decoupling decomposition from linear order.

- **Grammar encoding.** The encoding scheme for MGs defined in Section 3.2 and utilized throughout the dissertation is straightforward but naive: each LI is treated as a sequence of symbols from the same encoding table. For the most part, this is sufficient to produce results that are intuitively correct. However, the cost associated with longer string components becomes (relatively) lower as more syntactic features are added to *Base*. As we have seen in Section 6.4, this may lead to the algorithm failing to unify what is clearly multiple instances of the same morpheme. Treating a lexical item as a string of phonological segments (approximated by orthography) followed by syntactic features is explicitly a simplification – which, for instance, completely ignores the semantic component. This is a limitation which can potentially be lifted by selecting a different encoding scheme. Further refinements may be put in place to encourage higher level generalizations – for example, to lower the cost of reusing existing syntactic feature bundles or type sequences.

More fundamentally, the next step is to improve the evaluation measure. Using a grammar formalism and a quantitative metric is not guaranteed to produce results that are in line with how linguists think of language structures; see e.g. de Marcken (1995) on stochastic context-free grammars misinterpreting English phrase structure. In our case, there is no gold standard for correct results, as long as one of our goals is to compare competing analyses. Furthermore, in Chapter 6 we have seen quite a few discrepancies between intuition and optimized results, some of which could be attributed to the limitations of the encoding scheme, the algorithm implementation, or the grammar formalism itself. However, there is a clear correlation between grammars with lower MDL values and those that appear linguistically plausible, which means that the approach reported here likely draws on the same information linguists use to make their generalizations.

Linguists put a lot of emphasis on obtaining independent evidence for their proposals to justify the theoretical cost of postulating a new structure or operation. In an informal setting, this evidence would be brought in as a set of examples. If the proposal is translated into a formal grammar, reusing a lexical item in multiple structures would translate into a quantifiable reduction to the grammar cost. With a sufficiently comprehensive grammar fragment and a good evaluation measure, encoding cost can keep track of the strength of every relevant argument and counter-argument. Theoretical devices can be factored into this cost much in the same way as language data. For example, an analysis of English syntax that made no use of covert movement might be cheaper to encode, because it would require fewer syntactic feature types and therefore fewer distinct symbols to encode; or its cost might be higher due to additional lexical items one might need to compensate for the lack of covert movement.

This dissertation started with a rather general question: how do linguists construct analyses of syntactic phenomena? Using it as a starting point, I have made a step towards formalizing the notion of *intuitive goodness* of syntactic descriptions and connecting it to the more easily definable notion of *quantitative goodness*. The next goal is to close the gap.

References

D. Adger. A minimalist theory of feature structure. *Features: Perspectives on a key notion in linguistics*, pages 185–218, 2010.

K. Arregi and A. Pietraszko. Generalized head movement. *Proceedings of the Linguistic Society of America*, 3(1):1–15, 2018.

M. Brody. Mirror theory: Syntactic representation in perfect syntax. *Linguistic Inquiry*, 31 (1):29–56, 2000.

N. Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

N. Chomsky. *Syntactic structures*. Mouton, The Hague, 1957.

N. Chomsky. Aspects of the theory of syntax. *MIT Press*, 1965.

N. Chomsky. *Knowledge of language: Its nature, origin, and use*. Greenwood Publishing Group, 1986.

N. Chomsky. *The Minimalist Program*. MIT Press, Cambridge, MA, 1995.

N. Chomsky. Minimalist Inquiries: the framework. In R. Martin, D. Michaels, and J. Uriagereka, editors, *Step by Step: Essays on Minimalist Syntax in Honor of Howard Lasnik*, pages 89–156. MIT Press, Cambridge, MA, 2000.

N. Chomsky. Derivation by phase. In M. J. Kenstowicz, editor, *Ken Hale: A Life in Language*, pages 1–52. MIT Press, 2001.

N. Chomsky and M. Halle. *The sound pattern of English*. Harper & Row New York, 1968.

A. Clark. Learning trees from strings: A strong learning algorithm for some context-free grammars. *Journal of Machine Learning Research*, 14:3537–3559, 2013.

A. Clark. Canonical context-free grammars and strong learning: two approaches. In *Proceedings of the 14th Meeting on the Mathematics of Language (MOL 2015)*, pages 99–111, 2015.

A. Clark. Computational learning of syntax. *Annual Review of Linguistics*, 3:107–123, 2017.

A. Clark and R. Eyraud. Polynomial identification in the limit of substitutable context-free languages. *Journal of Machine Learning Research*, 8(Aug):1725–1745, 2007.

C. de la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

C. de Marcken. Lexical heads, phrase structure and the induction of grammar. In *Third Workshop on Very Large Corpora*, 1995. URL https://www.aclweb.org/anthology/W95-0102.

D. Embick and A. Marantz. Architecture and blocking. *Linguistic inquiry*, 39(1):1–53, 2008.

M. Ermolaeva. Morphological agreement in minimalist grammars. In *International Conference on Formal Grammar*, pages 20–36. Springer, 2018.

M. Ermolaeva and D. Edmiston. Distributed morphology as a regular relation. *Proceedings of the Society for Computation in Linguistics*, 1(1):178–181, 2018.

M. Ermolaeva and G. M. Kobele. Agreement and word formation in syntax. Manuscript, 2019.

M. Fowlie. Order and optionality: Minimalist grammars with adjunction. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 12–20, 2013.

E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

E. M. Gold. Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

J. Goldsmith. Meaning and mechanism in grammar. *Harvard studies in syntax and semantics*, 3:423–449, 1980.

J. Goldsmith. Unsupervised learning of the morphology of a natural language. *Computational linguistics*, 27(2):153–198, 2001.

J. Goldsmith. An algorithm for the unsupervised learning of morphology. *Natural language engineering*, 12(4):353–372, 2006.

J. Goldsmith and F. Mpiranya. Learning Swahili morphology. In *Proceedings of the 49th Annual Conference on African Linguistics*, to appear.

J. Goldsmith, J. Lee, and A. Xanthos. Computational learning of morphology. *Annual Review of Linguistics*, 3:85–106, 2017.

T. Graf. *Local and Transderivational Constraints in Syntax and Semantics*. PhD thesis, UCLA, 2013.

J. T. Hale and E. P. Stabler. Strict deterministic aspects of minimalist grammars. In *International Conference on Logical Aspects of Computational Linguistics*, pages 162–176. Springer, 2005.

M. Halle and A. Marantz. Distributed morphology and the pieces of inflection. In K. Hale and S. Keyser, editors, *The view from building 20*, pages 111–176. The MIT Press, 1993.

H. Harley. Possession and the double object construction. *Linguistic variation yearbook*, 2 (1):31–70, 2002.

H. Harley and H. K. Jung. In support of the p_{HAVE} analysis of the double object construction. *Linguistic inquiry*, 46(4):703–730, 2015.

Z. S. Harris. *Methods in structural linguistics*. University of Chicago Press, 1951.

Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

Y. Hu, I. Matveeva, J. Goldsmith, and C. Sprague. Using morphology and syntax together in unsupervised learning. In *Proceedings of the Workshop on Psychocomputational Models of Human Language Acquisition*, pages 20–27. Association for Computational Linguistics, 2005.

S. Indurkhya. Automatic inference of minimalist grammars using an SMT-solver. *arXiv preprint arXiv:1905.02869*, 2019.

S. Indurkhya. Inferring minimalist grammars with an smt-solver. *Proceedings of the Society for Computation in Linguistics*, 3(1):476–479, 2020.

M. Johnson. Marr's levels and the minimalist program. *Psychonomic bulletin & review*, 24 (1):171–174, 2017.

M. Kanazawa. *Learnable Classes of Categorial Grammars*. PhD thesis, Stanford University, Stanford, CA, USA, 1995. UMI Order No. GAX94-29947.

R. Katzir. A cognitively plausible model for grammar induction. *Journal of Language Modelling*, 2, 2014.

M. Kawakami. Double object constructions: Against the small clause analysis. *Journal of Humanities and Social Sciences*, 45:209–226, 2018.

R. S. Kayne. *Connectedness and Binary Branching*. Foris, Dordrecht, 1984.

R. S. Kayne. *The antisymmetry of syntax*. MIT Press, 1994.

G. M. Kobele. Formalizing mirror theory. *Grammars*, 5(3):177–221, 2002.

G. M. Kobele. Lexical decomposition, 2018. *Computational Syntax* lecture notes.

G. M. Kobele. Minimalist grammars and decomposition. In K. K. Grohmann and E. Leivada, editors, *The Cambridge Handbook of Minimalism*. Cambridge University Press, Cambridge, to appear.

G. M. Kobele, T. Collier, C. Taylor, and E. P. Stabler. Learning mirror theory. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+ 6)*, pages 66–73, 2002.

G. M. Kobele, J. Riggle, T. C. Collier, Y. Lee, Y. Lin, Y. Yao, C. E. Taylor, and E. P. Stabler. Grounding as learning. In *Language Evolution and Computation Workshop, ESSLLI'03*, 2003.

R. K. Larson. On the double object construction. *Linguistic inquiry*, 19(3):335–391, 1988.

H. Lasnik. Verbal morphology: Syntactic structures meets the Minimalist Program. *Evolution and revolution in linguistic theory: Studies in honor of Carlos P. Otero*, pages 251–275, 1995.

S. Laszakovits. Case theory in minimalist grammars. In *International Conference on Formal Grammar*, pages 37–61. Springer, 2018.

A. Marantz. No escape from syntax: Don't try morphological analysis in the privacy of your own lexicon. *University of Pennsylvania working papers in linguistics*, 4(2):14, 1997.

D. Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. Henry Holt and Co., Inc., New York, NY, USA, 1982. ISBN 0716715678.

J. Merchant. Roots don't select, categorial heads do: lexical-selection of PPs may vary by category. *The Linguistic Review*, 36(3):325–341, 2019.

J. Michaelis. Derivational minimalism is mildly context–sensitive. In *International Conference on Logical Aspects of Computational Linguistics*, pages 179–198. Springer, 1998.

C. Peacocke. Explanation in computational psychology: Language, perception and level 1.5. *Mind & language*, 1(2):101–123, 1986.

D. Pesetsky. Zero syntax. vol. 2: Infinitives, 1991.

D. M. Pesetsky. *Zero syntax: Experiencers and cascades*. MIT press, 1996.

S. Pinker. *Language Learnability and Language Development*. Cambridge University Press, 1984.

S. Pinker. The bootstrapping problem in language acquisition. *Mechanisms of language acquisition*, pages 399–441, 1987.

J.-Y. Pollock. Verb movement, universal grammar, and the structure of ip. *Linguistic inquiry*, 20(3):365–424, 1989.

E. Rasin and R. Katzir. On evaluation metrics in optimality theory. *Linguistic Inquiry*, 47 (2):235–282, 2016.

E. Rasin, I. Berger, N. Lan, and R. Katzir. Learning phonological optionality and opacity from distributional evidence. In *Proceedings of NELS*, volume 48, pages 269–282, 2018.

R. Reddy. Speech understanding systems: summary of results of the five-year research effort at Carnegie-Mellon University., 1977.

J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

S. M. Shieber. Evidence against the context-freeness of natural language. In *Philosophy, Language, and Artificial Intelligence*, pages 79–89. Springer, 1985.

J. M. Siskind. A computational study of cross-situational techniques for learning word-to-meaning mappings. *Cognition*, 61(1-2):39–91, 1996.

E. P. Stabler. Derivational minimalism. In C. Retoré, editor, *Logical Aspects of Computational Linguistics: First International Conference, LACL '96 Nancy, France, September 23–25, 1996 Selected Papers*, pages 68–95. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

E. P. Stabler. Recognizing head movement. In *Proceedings of the 4th International Conference on Logical Aspects of Computational Linguistics*, LACL '01, pages 245–260, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42273-0.

E. P. Stabler and E. L. Keenan. Structural similarity within and among languages. *Theoretical Computer Science*, 293(2):345–363, 2003.

E. P. Stabler, T. C. Collier, G. M. Kobele, Y. Lee, Y. Lin, J. Riggle, Y. Yao, and C. E. Taylor. The learning and emergence of mildly context sensitive languages. In *European Conference on Artificial Life*, pages 525–534. Springer, 2003.

J. Torr and E. Stabler. Coordination in minimalist grammars: Excorporation and across the board (head) movement. In *Proceedings of the 12th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+ 12)*, pages 1–17, 2016.

E. S. Williams. Small clauses in English. In *Syntax and Semantics volume 4*, pages 249–273. Brill, 1975.

R. Yoshinaka. Efficient learning of multiple context-free languages with multidimensional substitutability from positive data. *Theoretical Computer Science*, 412(19):1821–1831, 2011.