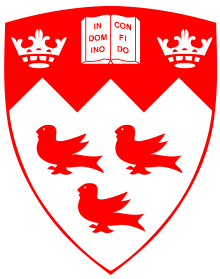
Automatic Melodic Analysis

Ryan Alexander Groves



Music Technology Area
Schulich School of Music
McGill University
Montreal, Canada

December 2015

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Arts.

Abstract

Melodic reduction is a process performed on symbolic music to discover the more important structural notes of a melody. In music theory literature the process has a foundation in the works of Heinrich Schenker, as well as Fred Lerdahl, and Ray Jackendoff. Among other applications, melodic reduction can be used for: harmony estimation, melodic similarity and comparison, compression of melodic representations, melodic search, and automatic or assisted composition. The process of melodic reduction can also involve auxiliary melodic analysis methods such as melodic segmentation (or grouping), metrical structure analysis, and melodic parallelism. This thesis investigates the use of a technique originally developed for Natural Language Processing (NLP) to identify hierarchies in sequential data—the Probabilistic Context-Free Grammar (PCFG).

Using the PCFG, it is possible to encode the rules of embellishment, allowing the identification of melodic embellishments. After defining each rule for a PCFG, it can take a sequence of notes and automatically build a tree structure by iteratively applying the rules. The hierarchical tree structure represents a reduction, since every subsequent level contains less and less notes. This thesis involves the supervised training of a PCFG using melodic reductions that were annotated using The Generative Theory of Tonal Music (GTTM) (Lerdahl and Jackendoff 1983). The GTTM dataset provides the melodic reductions in tree format, which is compatible with the PCFG technique (Hamanaka, Hirata, and Tojo 2007b). By leveraging the existing reductions, one can model the distributions for each of the rules contained in the analyses, and use those to find the most probable melodic reduction for a new input melody. A standard evaluation methodology is used to test the efficacy of the melodic reduction grammar.

Résumé

La réduction mélodique est un traitement réalisé sur la musique symbolique dans le but de découvrir les notes structurelles les plus importantes d'une mélodie. Dans la littérature de la théorie musicale, le traitement trouve son fondement dans les œuvres d'Heinrich Schenker, ainsi que de Fred Lerdahl et de Ray Jackendoff. Entre autres applications, la réduction mélodique peut être utilisée pour effectuer : Une estimation de l'harmonie, la similarité et la comparaison mélodiques, la compression de représentations mélodiques, la recherche mélodique, et la composition automatique ou assistée. Le processus de la réduction mélodique peut également impliquer des méthodes auxiliaires d'analyse mélodique, comme par exemple la segmentation (ou le regroupement) mélodiques, l'analyse de la structure métrique, et le parallélisme mélodique. Cette thèse étudie l'utilisation d'une technique originellement développées pour le traitement automatique du langage naturel (NLP) pour identifier les hiérarchies dans les données séquentielles La grammaire probabiliste indépendante du contexte (PCFG).

En utilisant la PCFG, il est possible d'encoder les règles d'embellissement, ce qui permet l'identification des embellissements mélodiques. Après avoir défini chaque règle pour un PCFG, elle peut prendre une séquence de notes et de construire automatiquement une structure arborescente par le itérativement application des règles. La structure hiérarchique arborescente représente une réduction, puisque chaque niveau suivant contient des notes de moins en moins. Cette thèse implique la formation supervisée de la PCFG en utilisant les réductions mélodiques qui ont été annotées en utilisant La Théorie Générative de la Musique Tonale (GTTM) (Lerdahl et Jackendoff 1983). Les fichiers des données GTTM présentent les réductions mélodiques sous forme d'arborescence, ce qui est compatible avec la PCFG technique (Hamanaka, Hirata, et Tojo 2007b). En tirant partie des réductions existantes, on peut modeler les distributions pour chacune des règles du dispositif de formation et les utiliser pour trouver l'arborescence la plus probable pour une nouvelle entrée de mélodie. Une méthodologie standard d'évaluation est utilisée pour tester l'efficacité de la grammaire de réduction mélodique.

Acknowledgements

First and foremost, I would like to thank my advisor, Ichiro Fujinaga. His guidance in both the technical challenges as well as his consistent support, feedback, and flexibility were invaluable to the project. I am also grateful for the malleability of the McGill Music Technology program, which allowed for the simultaneous pursuit of high-level Music Theory in addition to the application of cutting-edge Machine Learning techniques.

On a technical level this thesis was made possible by the open source Natural Language Toolkit (NLTK), as well as the database of expert musical reductions using The Generative Theory of Tonal Music (GTTM) that were provided to me by Masatoshi Hamanaka.

I would also like to thank my partner, Amanda, who supported me throughout the process with flights, Skypes, and a formidable tolerance for puns, and my family, who were as supportive as they were patient.

I am also indebted to my fellow labmates, first and foremost Hannah Robertson, who showed me how to be a Master's student, but also Greg Burlet, Andrew Hankinson, and Gabriel Vigliensoni who were always there to lend a helping hand.

Last but not least, I am thankful for the solidarity of my Music department brethren: Julian Vogels, Håkon Knutzen, James Perrella, and Emily Burt.

Contents

List of Figures

List of Tables

Glossary

CFG Context-Free Grammar. 1, 5, 7–11, 15–17

EFSC Essen Folksong Collection. 1

GPR Grouping Preference Rule. 1, 11

GTTM The Generative Theory of Tonal Music. 1, 11–14, 16

IOI inter-onset interval. 1

LBDM Local Boundary Detection Model. 1

MPR Metrical Preference Rule. 1, 11

NLP Natural Language Processing. 1

NLTK Natural Language Toolkit. 1, 17

OOI offset-to-onset interval. 1

PCFG Probabilistic Context-Free Grammar. 1, 5, 7, 9, 11, 12, 17

PRPR Prolongational Reduction Preference Rule. 1, 11

PRWFR Prolongational Reduction Well-Formedness Rule. 1, 12

TSPR Time-Span Preference Rule. 1, 11

TSWFR Time-Span Well-Formedness Rule. 1, 12

XML Extensible Markup Language. 1, 11, 12

1. Introduction and Motivation

When experiencing music, a listener engages in a combination of perceptual processes, from the translation of oscillating vibrations into pitch, to the automatic inference of more structural musical elements such as key and meter. Often, a music listener considers a melody to be a sequence of pitched events. However, when a listener hears a simple sequence of pitched events, they are experiencing more than just that particular sequence—they are forming a structural representation of that melody based on musical relationships within the sequence. For this reason, a listener can recognize a melody if it begins on a different pitch, in a different key (such as the "Happy Birthday" song started from an arbitrary pitch). Indeed, our perception of melodies often involves more than just the melody itself.

Musical structure is inherently hierarchical. In most songs, there is some form of repetition (unless the song is through-composed), which creates a particular high-level form of the piece. In Western tonal music, for example, there is the sonata form, which consists of three parts: an exposition, development, and a recapitulation. In popular music, songs can be broken into sections such as the intro, verse, and chorus (among others). In jazz music, performers often notate songs using a musical chart that contains each section with the chord sequence and duration for each chord in that section. In all of these forms, certain sections repeat in full or in part. Beyond the section structure, music continues to separate into musical objects that can be associated hierarchically; sections will contain phrases, which contain harmonic sequences, which will in turn contain notes.

Melodic reduction is the process of determining the more structural notes in a melody. During this process, a musical analyst will systematically remove notes from the melody that are deemed less structurally important. Some reasons for removing a particular note are, among others, pitch placement, metrical strength, surrounding notes, and relationship to the underlying harmony. Because of its complexity, formal theories on melodic reduction that comprehensively define each step required to reduce a piece in its entirety are relatively few.

Composers have long used melodic ornamentations to elaborate certain notes, or to span a particular interval. In the early 1900s, Heinrich Schenker developed a hierarchical theory of music reduction (a comprehensive list of Schenker's publications was assembled by David Beach (1969)). In it, he ascribed each note in the musical surface as an elaboration of a representative musical object found in the deeper levels of reduction. As part of the analysis, Schenker described particular methods of ornamentation that can also be used for the opposite operation—what he named *diminution* (Forte and Gilbert 1982). The particular categories of ornamentation that were used in his reductive analysis were *neighbor tones*, *passing tones*, *repetitions*, *consonant skips*, and *arpeggiations*. For example, given an interval formed by notes of longer durational

value, one can create an elaboration of that interval using additional notes of smaller value that conform to these categories of ornamentation. Similarly, if the particular ornamentation is identified, an analyst can remove certain smaller-valued notes so that only the larger-valued notes remain.

In the 1980s, Fred Lerdahl and Ray Jackendoff—a musician and a linguist—created a new theory of musical reduction in the The Generative Theory of Tonal Music (GTTM) (Lerdahl and Jackendoff 1983). The authors' goal was to create a formally-defined generative grammar for identifying the hierarchical structure of a musical piece. In GTTM, every musical object in a piece is *subsumed* by another musical object, which means that the subsumed musical object is directly subordinate to the other. Unlike Schenkerian analysis, this subordination rule necessitates that every event is accounted for by another single musical event. In detailing this process, Lerdahl and Jackendoff begin with breaking down metrical hierarchy, then move on to identifying a grouping hierarchy (separate from the metrical hierarchy). Finally, they create two forms of musical reductions using the information from the metrical and grouping hierarchies—the time-span reduction, and the prolongational reduction. The former details the large-scale grouping of a piece, while the latter notates the ebb and flow of musical tension in a piece. Between all of these sub-tasks, the particulars of their reductive process involve other analytical methods, such as parallelism, musical segmentation, cadence identification, and harmonic ordering/precedence. It is useful, then, to review the details of GTTM in order to understand all of the related music analytical techniques that might influence melodic reduction.

The focus of this thesis is to test the efficacy of the Probabilistic Context-Free Grammar (PCFG) when applied to the task of melodic reduction. The PCFG represents a generative grammar—much the same as the theoretical model that Lerdahl and Jackendoff aimed to create (Lerdahl and Jackendoff 1983). The additional feature provided by the PCFG is that each grammatical rule is also assigned a probability, each of which can be learned from a data set. The technique is borrowed from Natural Language Processing (NLP), and has been used before to perform the task of melodic reduction (Gilbert and Conklin 2007). However, in previous research, unsupervised learning was used, and the results were not tested against a dataset. In this thesis, the PCFG will be used to encode the rules of ornamentation into grammatical production rules, and the training process of the PCFG will use supervised learning to discover the probabilities associated with each rule. The grammar will be implemented using the open-source Natural Language Toolkit (NLTK) (Loper and Bird 2002), which provides methods for supervised learning of PCFGs. Once trained, the algorithm can determine the most probable parse tree for an arbitrary input melody. For supervised learning of a PCFG, it is necessary to have existing solutions in order to determine the probabilities assigned to each production rule in the grammar. In the process of implementing GTTM in software, Hamanaka et al. created such a database, with 300 expert analyses of melodies using the rules of GTTM (Hamanaka, Hirata, and Tojo 2007b). This dataset will be split into five separate folds. Then, cross-fold validation (Jurafsky and Martin 2000, 154) will be performed to evaluate the musical effectiveness of the melodic reduction technique. The results will be analyzed and discussed.

Motivations for performing melodic reduction include melodic identification and similarity, efficient storage of melodies, automatic composition, variation matching, and automatic harmonic analysis. For example, Rizo developed a method for symbolic music comparison utilizing tree data structures, which can compare melodies with different ornamentations (Rizo 2010). Marsden explored the use of Schenkerian reductions for identifying variations of melodies (Marsden 2010). De la Puente, Alfonso, and Moreno created a method for discovering the tree structures of melodies and subsequently evolving them using evolutionary grammars for automatic composition (2002).

1.1. Thesis structure

Rather than cover the entire field of melodic analysis techniques, which is too broad, the literature review for this thesis will focus on the melodic analysis techniques that utilize generative grammars and tree data structures. This focus on tree data structures was inspired by Marsden (2005), in which he discussed the ideal characteristics of the representation of musical structure. In order to represent musical structure, the author argued that an encoding must be both generative and hierarchical, among others. Marsden recommended a modified tree structure. Indeed, the abstract data type tree seems the most fit for capturing both the hierarchical and temporal aspects of a piece.

The thesis is organized as follows:

Chapter 2 provides background information on both GTTM and formal grammar theory, including probabilistic techniques. The background on GTTM will give an overview of the different musical theories that are required to perform a reduction, while the grammatical background will cover the computer science techniques that were used to perform melodic reduction with a PCFG.

Chapter 3 presents a history of literature on melodic reduction and the related methods that are required when trying to find melodic structure from symbolic music.

Chapter 4 explains the PCFG for melodic reduction algorithm and how it is trained.

Chapter 5 describes the process of evaluating the performance of the PCFG algorithm.

Chapter 6 discusses the results of the evaluation of the PCFG, as compared with GTTM.

Chapter 7 suggests future work to improve and extend the current algorithm.

2. Background

Before introducing the range of research on melodic reduction and its related techniques, a deeper look at certain theories and techniques utilized in this thesis is in order. It is first important to understand the mechanics of a Probabilistic Context-Free Grammar (PCFG). Secondly, since the dataset that is used for supervised learning was created using The Generative Theory of Tonal Music (GTTM), it is also important to understand the details of the form and content that is embedded in that theory. Unravelling the different layers of GTTM can also provide insight into which particular musical analysis methods are related to melodic reduction. This chapter will give an overview of the fundamental theories involved with Natural Language Processing (NLP), as well as a more in depth look into the formative concepts of GTTM.

2.1. Natural Language Processing

2.1.1. Formal Grammars

Grammars are effective tools for breaking down the hierarchical structure of a sequence. They were created to perform the grammatical extraction of written sentences; using a series of rewritable rules, they are able to describe the different possibilities of grammatical syntax. The field that grew around grammars is called NLP. Chomsky formalized grammars (1956) and later extended the theory (1959). Backus et al. extended the theory as well, in parallel (1959). The definition of a formal grammar consists of four parameters, $G = \{N, \Sigma, R, S\}$, which are defined as follows (Jurafsky and Martin 2000):

N a set of non-terminal symbols

Σ a set of terminals (disjoint from N)

R a set of production rules, each of the form $\alpha \rightarrow \beta$

S a designated start symbol

To be clear, production rules define the relationship between the non-terminals and the terminals; they specify which symbols can be replaced, or rewritten by other symbols. The non-terminal symbols, N, are equivalent to variables that can be rewritten by as any sequence of non-terminals or terminals. It is the production rules that specify these rewrites. By convention, non-terminals are represented with uppercase letters, and lowercase letters represent terminals. Within each production rule, both the α and the β represent a sequence of non-terminals and terminals. This is denoted by $\alpha, \beta \in \{\Sigma \cup N\}^*$,

where $\{\Sigma \cup N\}^*$ is the infinite set of strings of non-terminals and terminals. This set may or may not include the empty string, λ. Terminals are often called "words" because grammars were originally designed to parse written natural languages. A sequence of words creates a "sentence". These terms are often still used, even when using grammars for purposes other than written natural language (e.g., computer languages).

Each formal grammar provides all of the information needed to generate a single formal language. For this reason, formal grammars are synonymous with generative grammars. Similarly, formal grammars are descriptive grammars because they define the syntax of a language.

2.1.2. Parsing Algorithms

The process of applying a grammar to a given sentence in order to create a tree of applied production rules is called parsing. It is possible that a given grammar cannot describe an input sentence, or even that multiple parse trees result; when multiple trees can result, the grammar is known as *ambiguous*. Because of the recursive nature of grammars, a record of which rules have been applied to particular sub-strings is often kept, in order to avoid processing the same substring more than once. The record is known as a chart, and, subsequently, grammar-parsing algorithms are known as chart parsing algorithms (Kay 1986).

There are two types of parsing algorithms: bottom-up parsers or top-down parsers. Bottom-up parsers apply the rules that described the terminals first, and then build the tree up by applying rules to those non-terminals, while top-down parsers apply the rules with non-terminals first and search for possible terminals last. The predominant bottom-up parser is named after its co-creators, the Cocke-Younger-Kasami algorithm (Cocke 1969; Younger 1967; Kasami 1965).

2.1.3. Chomsky Hierarchy

Chomsky detailed the different types of formal grammars, and how they are related (Chomsky 1956). All formal grammars are related, and stem from the formal grammar as described above. The way they differ is dependent on the restrictions that are applied to the generic formal grammar. The most important distinction is the concept of productions that require or do not require *context*. In formal grammars, the non-terminals are considered variables. If these variables can only be applied with the requirement of adjoining terminals, then the grammar is context-sensitive—that is, if any of the rules contain terminals as well as non-terminals on the left-hand side. Thus, in Context-Sensitive Grammars, the syntax for a production can be

$$\begin{aligned} xA &\rightarrow \beta \\ Ay &\rightarrow \beta \\ xAy &\rightarrow \beta \end{aligned} \tag{2.1}$$

In these examples, A represents a non-terminal, while x and y represent terminals. The β symbol represents a sequence that includes either non-terminals, terminals, or both.

Context-Sensitive Grammars have a further requirement that, for any production rule, the number of symbols on the left-hand side must be less than or equal to the number of symbols on the right-hand side: $|\alpha| \leq |\beta|$.

Context-Free Grammars (CFGs) are just the opposite. CFGs require that the left-hand side of any production rule include only a single non-terminal. This ensures that the non-terminals can be applied regardless of the symbols surrounding it. The format for productions of CFGs is always:

$$A \rightarrow \beta \tag{2.2}$$

Regular grammars further restrict the format of productions. The right-hand side of any production that is part of a regular grammar must consist of any number of terminals, optionally succeeded by a single non-terminal. Alternatively, the order can be switched, such that the right-hand side contains any number of terminals preceded by a single non-terminal. It is also possible to make a rule with only terminals on the right-hand side. These first two options are described as right-linear (where the non-terminal is on the right) and left-linear. Every *regular* language can be produced by both a left-linear and an equivalent right-linear grammar. The left-hand side of each production, as in CFGs, must be composed of a single non-terminal. The family of languages produced by regular grammars can be obtained using regular expressions.

These cumulative restrictions allow the definition of language types that have a different level of *generative power*. Generative power is the ability to produce a language of a certain complexity. With each added restriction, the generative power of a grammar type wanes. For instance, a CFG can describe a language that a regular grammar simply cannot.

All of these types of grammars combined compose the Chomsky Hierarchy (Chomsky 1956). Each grammar type is set-inclusive, such that, for example, a regular grammar is a CFG, a CFG is a Context-Sensitive Grammar, and a Context-Sensitive Grammar is a formal grammar (see Figure 2.1).

2.1.4. Chomsky Normal Form

Chomsky also created a specific format for simplifying formal grammars. Chomsky Normal Form (CNF) requires that a grammar have rules with only two possible formats (Chomsky 1959):

$$A \rightarrow B \ C \tag{2.3}$$

or

$$A \rightarrow x \tag{2.4}$$

By definition, a grammar that is in CNF is also a CFG, since there are only non-terminals on the left-hand side of each rule. Notice, also, that a grammar in CNF must exclude the use of the empty string. Similarly, any CFG can be converted into CNF in an efficient way, by adding and substituting rules. For example, consider a CFG that has

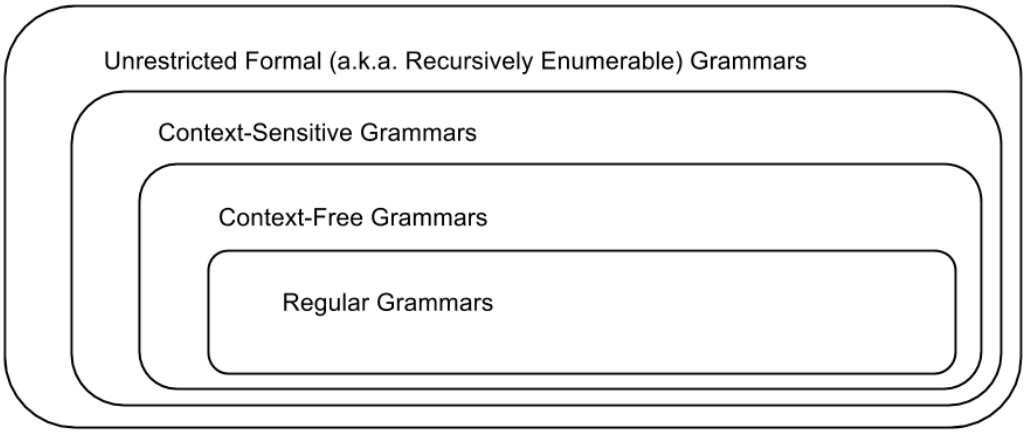


Figure 2.1.: The set of all possible grammars, in order of decreasing generative power.

the following set of productions:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow B\, x\, C \\ B &\rightarrow y \\ C &\rightarrow z \end{aligned} \tag{2.5}$$

The rule $A \rightarrow B\, x\, C$ is the only rule that prevents this CFG from being in CNF. In order to convert this CFG into its grammar equivalent in CNF, one must change the rule $A \rightarrow B\, x\, C$, and an additional two rules must be added:

$$\begin{aligned} A &\rightarrow B\ D \\ D &\rightarrow E\ C \\ E &\rightarrow x \\ B &\rightarrow y \\ C &\rightarrow z \end{aligned} \tag{2.6}$$

This new grammar is equivalent to the original one because it generates exactly the same language. However, it now has the added benefit that it is in CNF.

A grammar that is in CNF has certain desirable properties. First, the parse tree for any string of length n contains exactly 2^{n-1} nodes. This allows for efficient parsing of strings. In fact, the Cocke-Younger-Kasami algorithm, described below, was designed specifically to parse CFGs in CNF (Cocke 1969; Younger 1967; Kasami 1965). Second, there is no ambiguity—a single unique parse tree generates each string in the language.

2.1.5. Probabilistic Context-Free Grammars

An extension of the CFG is the PCFG (Booth 1969). Sometimes called the Stochastic Context-Free Grammar, the PCFG's definition is identical to the CFG, except for one addition to each rule. The rules are now defined as (Jurafsky and Martin 2000):

$$A \rightarrow \beta \ [p] \tag{2.7}$$

where p is the conditional probability of this particular non-terminal A expanding into the right-hand side β. p can also be written as $P(A \rightarrow \beta|A)$ or simply $P(A \rightarrow \beta)$. For each non-terminal, the conditional probabilities for the expansion of every rule that contains it all sum to 1:

$$\sum P(A \rightarrow \beta) = 1 \tag{2.8}$$

One can also compute the probability of a parse tree, given an input sentence S. The probability of a given parse tree, T, can be computed by considering every n non-terminal node in the tree, and cumulatively multiplying the probability associated with the particular production that the non-terminal used:

$$P(T, S) = \prod_{i=1}^{n} P(A_i \rightarrow \beta_i) \tag{2.9}$$

Because grammars can often be ambiguous, having two or more valid parse trees for the same input sentence, the PCFG can be used to disambiguate between the two. For any given sentence, the probability of each parse can be computed. A reasonable choice for the best parse is simply to choose the parse of the highest probability.

2.1.6. Training of a PCFG

A PCFG is only useful if the probabilities have been defined for each rule. The simplest way to gather the statistics for the likelihood of each rule application is to compute the percentages from an existing treebank. A treebank is a corpus in which every sentence has been annotated with its corresponding tree parse (Jurafsky and Martin 2000, 404). Given a set of sentences with their solution trees, it is simply a matter of adding up the occurrences of each rule and dividing by the number of times a specific right-hand side expansion occurs. This yields probabilities for every expansion, and is known as "inducing" a PCFG. It is a form of supervised training.

This thesis will follow a similar approach. Researchers have previously created a corpus treebank of 300 melodic analyses in which there exists a tree structure for each melody (Hamanaka, Hirata, and Tojo 2007b). In this thesis, a system is designed to convert the tree structures provided in that treebank into the appropriate format for the training of a PCFG, and the resulting probabilities will allow the most likely parse tree to be found for any new melody.

2.2. The Generative Theory of Tonal Music

The Generative Theory of Tonal Music (GTTM) was the product of musician Lerdahl and linguist Jackendoff, who attempted to use linguistic ideas to determine the hierarchical musical structure of a piece (including both the rhythm and pitch) (1983). Lerdahl and Jackendoff focused on the perception of music by the listener, in terms of the musical structure that is communicated aurally. Indeed, they believe that "a piece of music is a mentally constructed entity, of which scores and performances are partial representations by which the piece is transmitted"(Lerdahl and Jackendoff 1983, 7). Thus, they believe it is imperative to consider the cognitive processes involved with processing music, and to incorporate those into any theory of musical structure. Lerdahl and Jackendoff sought to encode the psychological principles of the Gestalt tradition—specifically the works of Wertheimer (1938), Köhler (1929), and Koffka (1935)—into grammatical musical rulesets. The motivation for their research was to provide a grammar that is machine-executable, however they noted that their focus does not lie in formally defining the precise mathematical formulae needed to execute the grammar (Lerdahl and Jackendoff 1983, 53). It is important, then, to consider which aspects of their formal descriptions are the most challenging to implement as a mathematical, or software system.

Before delving into the details of GTTM, a general overview of the different facets of the theory is in order. The decisions made for the representation and structure of the theory play a large role in the manifestation of the rules.

Following that, a presentation of the motivations and intuitions of the rules that form each of the four components will be detailed. With each of these components, another facet of the process of identifying the musical structure will be evident. Given the tree-based form of the resulting musical structure, one can consider the results as a musical reduction. In the context of building a PCFG for melodic reduction, it is also important to understand, in detail, previous attempts at building generative grammars for musical reduction. The different features of GTTM could be used, for example, to form the production rules of the PCFG. Similarly, the dataset that will be used in this research for training the PCFG was created using GTTM. In order to understand what branching decisions were made in the training data, one must understand the principles upon which the musical reduction trees were formed.

2.2.1. The Four Components

GTTM is comprised of four separately defined, yet interacting components. The set consists of four formally defined hierarchical organizations: metrical structure, grouping structure, time-span reduction, and prolongational reduction. Grouping addresses the organizing of similar gestures on the musical surface, as well as the relationship between larger groups of notes. The metrical structure creates a hierarchy of strong and weak beat associations. The time-span reduction combines both meter and grouping to create a segmented reduction that pinpoints the strongest events in each segment (called structural accents), while the prolongational reduction creates a hierarchy of structural

pitch content organized by the ebb and flow of tension in the piece. Each of these are annotated as a hierarchy containing different types of elements that are specific to that type of analysis, however the hierarchies share similar properties. For each type of hierarchy, the following formal definitions apply:

1. It is "composed of discrete elements or regions related in such a way that one element subsumes or contains other elements or regions"(Lerdahl and Jackendoff 1983, 13).

2. No elements can overlap.

3. The combination of sub-elements for a given element must compose the entire element (there cannot be partial sub-elements, or elements that are not included).

4. It is recursive, such that the rules for an element and its sub-elements apply at every level.

5. Elements at the same hierarchical level must be contiguous (non-adjacent groupings are prohibited).

The elements that a hierarchy is composed of can be either single nodes that represent timed and pitched events, or the elements can be regions of a specific duration. In each case, the element that subsumes another element must have a higher priority in the musical hierarchy. For example, an element of higher priority could be one that has a larger duration or more prominent metrical placement. In some cases, there is reason to have overlap between adjacent elements, for instance, when the end of a cadence is also used as the starting note in a new phrase. The authors address these situations as exceptions to the rule, and create rules for duplicating an event so that it may exist as part of two separate elements.

Meter versus Grouping

Lerdahl and Jackendoff made a clear distinction between the rules necessary to describe meter, and the rules necessary to describe groups. Their reasoning was that groups should represent a sequence of related notes—a durational unit—while meter should represent just the onsets in time. Thus, the two systems for annotating the respective metrical and grouping structures of a piece should also be distinct. The notable counter-example that the authors provided is the representation of meter by prosodic analogy (Cooper and Meyer 1960). Cooper and Meyer presented a rhythm notation system that borrows its syntax from stress markers in poetry. Every element in the rhythmic structure is represented by a dash ('–') or a cup ('∪') symbol, to represent strong and weak beats, respectively. These symbols are used to notate both meter and grouping. There are disadvantages in the prosodic representation that Cooper and Meyer presented, as compared with Lerdahl and Jackendoff's. Generally, on the lowest level, the system will align with metrical accents, which is congruent with Lerdahl and Jackendoff's method of metrical notation. In higher levels of the notation, the notation

system is ambiguous as to whether it notates the meter or the grouping, and thus the system breaks down. Furthermore, the beats at the smaller level are not represented by specific points in time, rather they indicate a time-span. Lerdahl and Jackendoff argued that any metrical notation should be able to indicate a precise moment in time, similar to the conductor accenting imaginary, and infinitesimally small points of time with gestures. By identifying the drawbacks in prosodic notation, Lerdahl and Jackendoff were able to create a more accurate depiction of two musical features (meter and grouping) as separate systems.

2.2.2. Metrical Structure

Lerdahl and Jackendoff used a simpler approach than prosodic notation, by utilizing a dot notation for meter (see Figure 2.2), and also by separately defining the metrical and grouping notational systems. Meter is notated with sequences of dots stacked in levels, in which the weaker beats are not represented at a higher level. According to Lerdahl and Jackendoff: "the time-spans between beats at any given level must be either two or three times longer than the time-spans between beats at the next smaller level"(Lerdahl and Jackendoff 1983, 20). The authors also noted that not all metrical levels are equally perceptible by listeners. Lerdahl and Jackendoff argued that "the listener tends to focus primarily on one (or two) intermediate level(s) in which the beats pass by at a moderate rate. This is the level at which the conductor waves his baton, [and] the listener taps his foot."(Lerdahl and Jackendoff 1983, 21). Lerdahl and Jackendoff borrow a term from the Renaissance, and labels this unit from the metrical hierarchy as the *tactus*. The *tactus* is defined as the principal rhythmic unit (Sadie and Grove 1980). The *tactus* is generally continuous throughout the course of a piece. Levels smaller than the tactus, however, are allowed to enter and exit the piece as necessary. The principles just listed relate to what are referred to in GTTM as "Well-formedness Rules", which define how a hierarchical structure can be formed for the specific component. There can be many possible structures for a given musical excerpt, and the method by which these potential structures are rated is by a set of "Preference Rules". The Preference Rules are created with an order of priority, so that when two rules conflict, preference is given to one over the other.

For the metrical preference rule system, one particular rule establishes the logic for the majority of the Metrical Preference Rules (MPRs)[1]. MPR 3 states (Lerdahl and Jackendoff 1983, 76):

> Prefer a metrical structure in which beats of level L_i that coincide with the inception of pitch-events are strong beats of L_i.

Inceptions can be considered onsets in this context. The remaining MPRs extend the principle presented in MPR 3 to describe those salient attributes that might be important, such as: accent, long duration of a pitch-event, long duration of dynamic, long slurs, long patterns of articulation, long regions of identical pitch, and long duration

[1] A list of all the rules of GTTM can be found in Appendix A

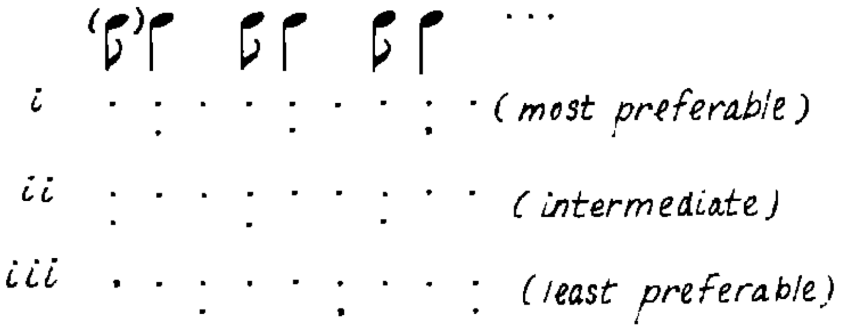


Figure 2.2.: An example metrical grid assignment based on the longer duration of certain notes in the musical snippet. Here, (*i*) is most preferable because the strong beats fall on the notes of longest duration. In both (*ii*) and (*iii*) the strong beats fall on notes of lesser length, and moments with no onset, respectively, and are therefore less preferable (Lerdahl and Jackendoff 1983, 80).

of the underlying harmonic unit. These preference rules provide a means by which one can narrow down the set of possible metrical structures given the pitch content, as well as the harmonic context of a piece. In many cases, notes of stronger metrical position are considered more important for the overarching structure of a piece.

2.2.3. Grouping Structure

According to Lerdahl and Jackendoff, Gestalt theory provides the foundation of the motivation for the set of rules that define the way in which notes should be grouped (Lerdahl and Jackendoff 1983, 36). It is useful to investigate with more detail the visual corollaries that were used to create the grouping rules of GTTM. First, the concept of grouping based on proximity is addressed in GTTM. If a person sees a group of three identical objects, spaced unevenly, he or she will naturally divide the group into two subgroups; one subgroup will consist of the two objects that are closer in distance, and the other subgroup will consist of the third, single object. Given identical objects in music—for example, three notes of identical duration—those notes that have a shorter distance between their onsets will be grouped together. Figures 2.3 and 2.4 show examples of the Gestalt rules applied to vision and music, respectively. The two circles on the left of Figure 2.3a can be cleanly grouped, while 2.3b shows a less certain grouping, and 2.3c cannot be grouped at all. Similarly, the examples 2.4a and 2.4b can be easily grouped, while 2.4c, 2.4d, and 2.4e show progressively less-defined grouping distinctions.

Lerdahl and Jackendoff borrow another principle from Gestalt grouping theory, which is the concept of grouping similar objects. In the visual analog, for example, in a group of objects that consists of two squares and one circle, a viewer will perceive the two squares as being part of a group because of their physical similarity. In terms of musical objects

Figure 2.3.: Examples of visual groupings formed by Gestalt distance rules (Lerdahl and Jackendoff 1983, 40).

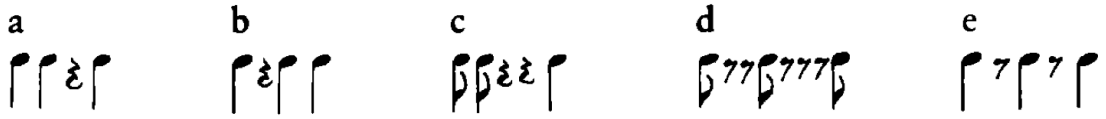


Figure 2.4.: Auditory analog of Gestalt distance-based grouping rules (Lerdahl and Jackendoff 1983, 40).

with identical duration, those notes that are closer in pitch will be aurally similar, and will therefore be grouped together. Much like the grouping principle of proximity, there is a range of ambiguity in the groupings (see Figure 2.5).

The grouping notation system is represented by slur symbols that span a certain amount of time, and is also hierarchical. The grouping system considers the differences between *transitions* of notes. To elaborate, take Grouping Preference Rule (GPR) 2, paraphrased here: Given a sequence of four notes $n_1n_2n_3n_4$, the transition n_2–n_3 may be considered a grouping boundary if either the offset-to-onset interval (OOI) or the inter-onset interval (IOI) are greater than its neighboring transitions (Lerdahl and Jackendoff 1983, 45). The OOI represents situations such as a break in slurring (i.e., both the neighbor transitions have a slur, and the current one does not) and rests. The IOI represents notes of differing duration (all other things equal). Duration is implicitly encoded in the detailing of these two separate parameters (since IOI minus OOI is equal to the duration of the first note in the pair). However, the rule for differing durations is also made explicit in the following preference rule, GPR 3. GPR 3 states that, when considering transitions in the same way as GPR 2, grouping boundaries will also form on transitions with change in register (intervallic distance), dynamics, articulation, and duration.

The rest of the GPRs deal with the organization of larger-level grouping. These rules are a lot less clear-cut, and involve more nuanced wording. Firstly, the larger-level preference rules state that the lower-level rules should be graduated to the higher levels

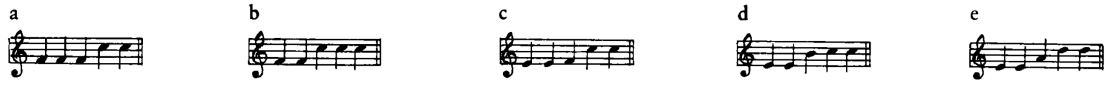


Figure 2.5.: Auditory analog of Gestalt similarity-based grouping rules. Figures ordered by increasing ambiguity (Lerdahl and Jackendoff 1983, 41).

when their effects are "relatively more pronounced", in GPR 4 (Lerdahl and Jackendoff 1983, 49). GPR 5 seeks to define the shape of higher levels, and states that higher levels should seek to subdivide the groups in two equal-length parts. GPR 6 requires that, when possible, parallel segments of music should form parallel groups, or parts of groups—the usage of a term like "parallel" is intrinsically complicated, as discussed below in Section 2.2.6. Finally, GPR 7 defines a preference rule that will prefer the stability of the time-span or prolongational reductions. Thus, there is a reference to preference rules from time-span and prolongational reductions.

Grouping is often included as part of the process of melodic reduction, and GTTM is no different. Section 3.1 will give a more detailed look at the research around melodic grouping. There are many that have formed their research around the same principles as those in GTTM and in Gestalt Psychology, while others provide different perspectives and methods.

2.2.4. Time-span Reduction

The two remaining components of GTTM that have yet been described are the time-span reduction, and the prolongational reduction. Both reductions are represented, in their totality, by a tree structure, as opposed to the dot or slur notation seen in the metrical and grouping hierarchies, respectively. The tree structures follow the formal guidelines for well-formed hierarchies, as mentioned previously, and extend it to designate a system of reductions within the structure.

Branching

In GTTM, a format for determining dominance between two objects in a tree of musical objects is presented. Each branch has a visual direction that specifies which object is the dominant one. Different branching possibilities are shown in Figure 2.6. In Figure 2.6h, for example, the branches are organized from most dominant on the left to least dominant on the right. The fundamental assumption behind the directional branching in the time-span and prolongational trees (defined below) is that each pitch event is a musical elaboration of another pitch event (Lerdahl and Jackendoff 1983, 113). Also, the elaboration (or, in reverse, reduction) of events is recursive in that higher levels will also be reduced to even higher levels, until only a single event exists. Given two pitch events, their representative tree will have two separate branches that connect at a point. For right-branching trees, the left branch will end at the connection point, while the right branch will continue upwards. This represents that the object at the end of the right branch is the dominant sub-element, and the left element is simply an elaboration of the dominating right element. For left-branching trees, the opposite is true. Furthermore, no branches are allow to cross, each element can only represent one branch, and each element must exist on the tree. See Figure 2.6 for a depiction of different tree possibilities.

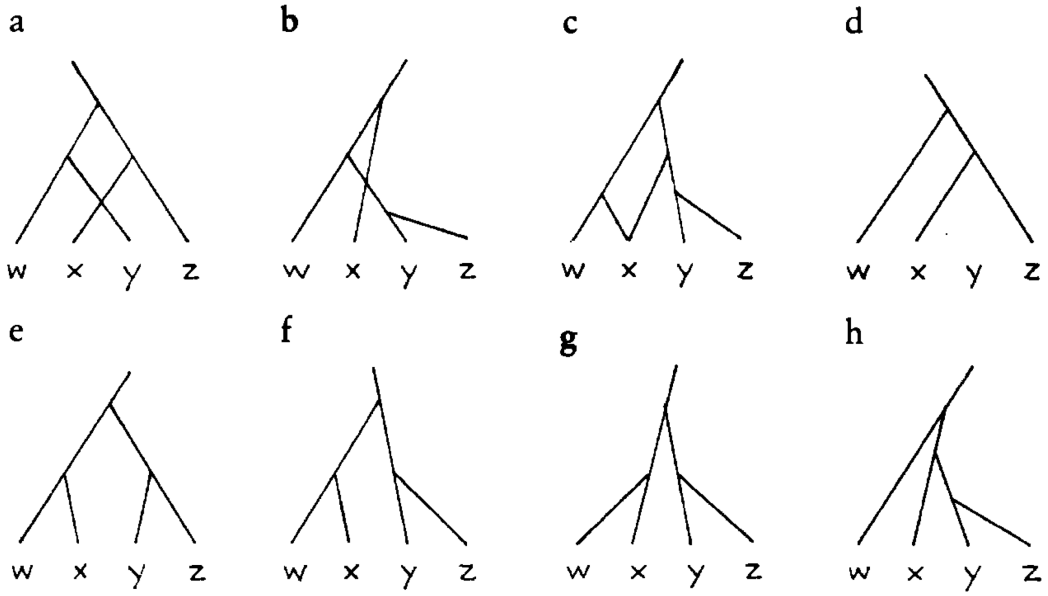


Figure 2.6.: Trees *a–d* represent invalid trees. *a* and *b* have overlapping branches, *c* has multiple branches for a single event (x), and *d* has no branch for event y. Trees *e–h* represent valid trees (Lerdahl and Jackendoff 1983, 114).

Time-span Reduction Preference Rules

The time-span reduction extends the grouping and meter components with an additional phrase grouping, as well as hierarchical pitch-based elements. It concerns itself specifically with reducing sets of vertical sonorities segmented by the metrical and grouping components. In general, vertical sonorities can be sets of multiple notes, such as chords, or single notes found in monophonic parts. Therefore, when considering which element subsumes an adjacent element, both rhythmic and harmonic principles must be considered.

Like the other components, GTTM lays out a set of preference rules that define the logic for parsing time-span reduction trees. There are two types of rules: local and non-local. Within these subgroups, the rules generally pertain to metrical considerations first, then harmonic considerations. For example, Time-Span Preference Rule (TSPR) 1 prescribes that the heads of each time-span should preferably be in a relatively strong metrical position, while TSRPR 2 prefers heads that are both relatively intrinsically consonant and relatively closely related to the local tonic. TSRPR 3 declares a preference for registral extremes, such that choices for the head of a time-span should weakly prefer a choice that has either a higher melodic pitch or a lower bass pitch. It is defined as a weak preference, so it will only be invoked when other preference rules are unavailable. The first three preference rules define the local considerations, and are generally prioritized over the following non-local rules (indeed, all the rules are ordered by preference). The non-local rules are similar; TSRPR 5 prioritizes branch heads that result in a more

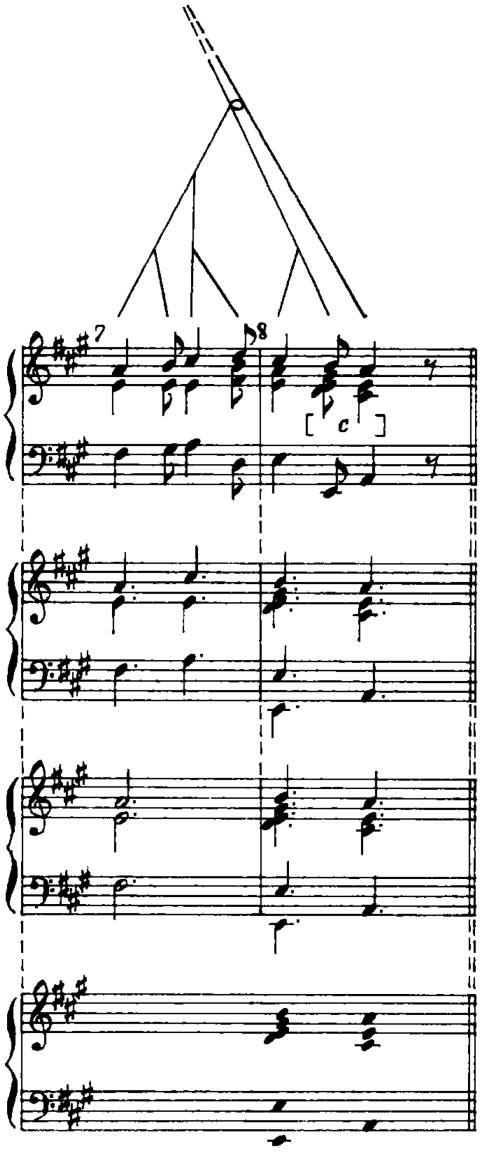


Figure 2.7.: An example of a notated cadence, marked with '[c]'.

stable choice of metrical structure, while TSRPR 6 creates a preference for heads that result in a more stable prolongational reduction (Lerdahl and Jackendoff 1983, 167). Here, there is a bit of ambiguity left in the definition because of the interdependence of the time-span reduction and the prolongational reduction. The authors were aware that conflicts would arise between the two, and that if there were such conflicts, the head that allows for more stability in the prolongational reduction following the time-span reduction, that head should have a priority over other possibilities. There is also a preference rule for Cadential Retention, TSRPR 7, and a couple of rules for defining the higher-level structure in terms of which types of heads should start or end the piece. With these explicit rules about cadences, it is evident that harmony is a guiding factor in the formation of prolongational trees, especially when considering the musical events higher in the tree. See Figure 2.7 for an example of the tree syntax for one instance of Cadential Retention.

Metrical Consonance

The time-span reduction's main organizational principles are based on meter (Lerdahl and Jackendoff 1983, 119). The time-span reduction of a piece notates and visualizes the structural accents and the general phrasing that a piece presents. In fact, at a local level (i.e., the musical surface), the time-span reduction is based largely on metrical considerations. With a time-span reduction, one can also identify parts with syncopation, where the local and the global structures conflict, as well as anacruses and afterbeats.

2.2.5. Prolongational Reduction

According to Lerdahl and Jackendoff, the time-span reduction is not designed to represent a listener's sense of progress in the piece; there is no way to encode the series of tensions and relaxations that a piece will usually invoke (Lerdahl and Jackendoff 1983, 179). The prolongational reduction seeks to define just that. The same format for directionally branched trees in the time-span prolongation applies to the prolongational reduction, however in this case the branches represent a hierarchy of tensing and relaxing events. A right-branching node implies that the child event on the right is more tense than the event on the left (which is represented at the next level as the head), and a left-branching node shows that there is a relaxation from the left child event to the right one.

Syntax

The prolongational reduction format also introduces some new branching elements. There are three additional specifications that warrant distinction with a new notation: strong prolongation, weak prolongation, and progression. Strong prolongation occurs when there are two musical events that have identical pitch information—they are the same. If these occur in functionally similar levels, the latter may be considered to prolong the first. Strong prolongations are notated by an empty circle at the point of branch intersection. Weak prolongation is similar, except the two musical events

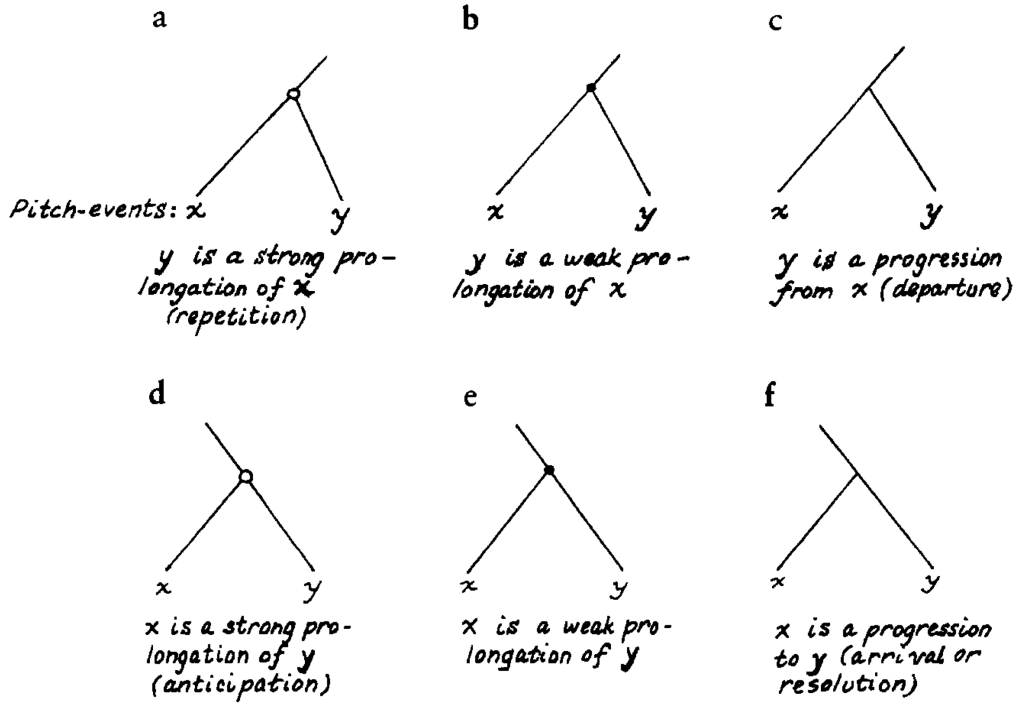


Figure 2.8.: A demonstration of the different branching possibilities for prolongational reduction (Lerdahl and Jackendoff 1983, 182).

do not have to be identical, they need only be similar. More specifically, "the roots of the two events are identical, but one of the events is in a less consonant position" (Lerdahl and Jackendoff 1983, 182). Dark, filled-in circles notate weak prolongations at the point of two intersecting branches. Lastly, the regularly-formed branches represent something slightly distinct from the earlier notations: a progression. This occurs when the harmonic roots of two subsequent events at some level are distinct. See Figure 2.8 for a depiction of the different directional branching for prolongational reduction.

Prolongational Form

The prolongational reduction follows a different approach than all of the other components. Mainly, the prolongational reduction tree for a piece is created from the top down. One starts with the time-span reduction, and assumes that all of the structural accents, and the most important events are segmented correctly. However, when the prolongational structure differs from the metrical and grouping structure, so will the prolongational tree differ from the time-span tree. The motivation for a top-down approach is that, in order to understand an event's prolongational function, one must be familiar with the events surrounding it at a higher level.

Since the method needs to be top-down, Lerdahl and Jackendoff presented a structural template that a piece can be expected to follow in Western tonal music, called the

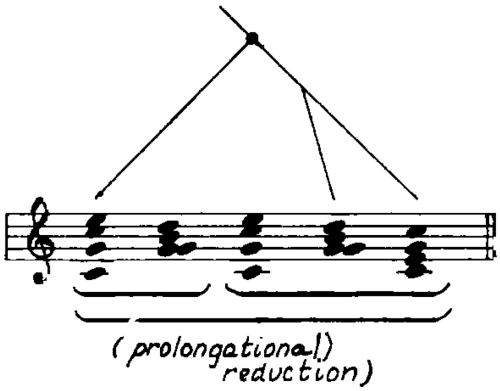


Figure 2.9.: Basic form for a prolongational reduction (Lerdahl and Jackendoff 1983, 189).

normative prolongational structure. From GTTM: "a tonal phrase or piece almost always begins in relative repose, builds toward tension, and relaxes into a resolving cadence" (Lerdahl and Jackendoff 1983, 198). With the *normative prolongational form*, Lerdahl and Jackendoff sought to specify the minimal branching criteria to encapsulate the flow of tension and release in tonal music. The authors made reference to the work of Heinrich Schenker, who theorized that all Western tonal music has the same underlying form, and created his own notation to describe this reduced form for any piece. However, Lerdahl and Jackendoff noted that Schenker's reductive levels do not form a hierarchy that satisfies the well-formedness constraints placed on prolongational reductions; one significant difference is that musical events are not necessarily subsumed by events at the next-highest level, which means that the function of some notes is left un-notated (Forte and Gilbert 1982).

The normative prolongational structure incorporates a basic form, which consists of a starting event, usually on the tonic, and ends with a cadence. The basic form is a recursive structure that can be applied over a whole song (as in the case of normative prolongational structure, or for a simple phrase on a much more local level). The final cadence is a weak prolongation of the initial tonic, and is notated as such (see Figure 2.9). Note that the initial event is a left-branching weak prolongation of the final event. The final event is also a progression from the middle event, denoting a relaxation. Normative prolongational structure must also include cadential preparation in the form of the subdominant (Lerdahl and Jackendoff 1983, 198). The subdominant is a common element in Western tonal music that precedes the dominant of the final cadence, and thus is formed as a left-branch in the prolongation reduction.

Prolongation Reduction Preference Rules

Prolongational reduction also contains preference rules for deciding which event in a particular time-span region is the most prolongationally important, and should thus be

the head of that region (Lerdahl and Jackendoff 1983, 220). First and foremost, there is a preference for the most important time-span event; if the time-span considers it the most important event in that region (often these are considered structural accents), then there is a strong preference to make that the prolongational head. This preference is defined in Prolongational Reduction Preference Rule (PRPR) 1. Similarly, the prolongational reduction should honor the segmentation that the time-span reduction created (PRPR 2). There is a preference for a more stable prolongational connection (PRPR 3). Here, there is an important clarification required, in order to define the stability condition. Before describing prolongational stability in more detail, it is wise to consider the final preference rules: Branches should also connect to more prolongationally stable events (PRPR 4); The normative form should be preferred when possible (PRPR 6).

Prolongational Stability

Lerdahl and Jackendoff stated that when making choices for the prolongational reduction, the branches should be maximally stable, based on the hierarchy of strong prolongations, weak prolongations and progressions (Lerdahl and Jackendoff 1983, 188). Right-branching preferences are for the prolongations in that order (strong, weak, progression), and left-branching preferences are the opposite. Beyond that, there is a preference for events with similar tonality. Events that have a common diatonic collection (e.g., those in the same key) are considered more stable. A lower, but still significant priority in the preference hierarchy is the melody. Melodies are branch based primarily on melodic distance—those notes that are closer in pitch will be more likely to be grouped together as branches. However, there are other concerns as well. For example, passing motion will generally attach to the event with stronger tonality, as defined by the explicit or implied harmony. Neighbor tones will generally right-branch, since they are usually a result of a strong prolongation, and strong prolongations are almost always right-branching. Metrical placement is also a consideration, but that should be taken care of by the time-span preference rules. The authors of GTTM also specify a preference for categorizing ascending melodic motion as a right-branching structure, and descending motion as a left-branching one. These branching structures are the more stable forms for each. Lastly, prolongational stability is defined by harmonic stability, in which two harmonic elements are deemed more stable if their harmonic roots are closer on the circle of fifths. Like melodies, progressions that ascend (on the circle of fifths) are right-branching and descending ones are left-branching.

2.2.6. Parallelism in GTTM

Parallelism plays a substantial role in the formulation of preference rules. The highest priority rule in the metrical preference rules (MPR 1) from The Generative Theory of Tonal Music (GTTM) is based on parallelism. The rule states “Where two or more groups or parts of groups can be construed as parallel, they preferably receive parallel metrical structure” (Lerdahl and Jackendoff 1983, 75). Furthermore, in the Grouping Preference Rules (GPRs), parallelism finds its own rule in GPR 6. It again occurs in the

Time-Span Preference Rules (TSPRs)—in TSPR 3—and the Prolongational Reduction Preference Rules (PRPRs)—in PRPR 5. However, the authors failed to provide a rule-based system for defining parallelism. The authors admit, "failure to flesh out the notion of parallelism is a serious gap in [their] attempt to formulate a fully explicit theory of musical understanding" (Lerdahl and Jackendoff 1983, 53).

3. History of Melodic Reduction and Related Techniques

The Generative Theory of Tonal Music (GTTM) is not the only theory that was created to formalize the music theoretical methods involved in melodic reduction. Schenkerian analysis (Beach 1969) also provides a framework with which to reduce scores. Beyond that, many concepts that are involved with the process of melodic reduction can be found in a history of research in the Music Information Retrieval (MIR) field, which will be explored in this chapter. These concepts include melodic segmentation, melodic parallelism, and the theory of harmonic progression. Each of these have been approached with different methods—some inspired by GTTM, and others that take a different approach. In addition, melodic reduction has been applied to automatic generation of melodies. Included in this chapter are methods for automatically reducing musical pieces using Schenkerian analysis, as well as the relevant research for related melodic reduction techniques, and an overview of generative tools for the automatic composition of melodies.

3.1. Melodic Segmentation

As was shown in the detailed analysis of GTTM in Chapter 2, melodic segmentation plays an important role in the reduction of melodies. In GTTM, melodic segmentation was detailed in the Grouping Preference Rules (GPRs) (see Section 2.2.3), and the two processes can be considered equivalent. The time-span reduction and the prolongational reduction tree are only formed after the grouping hierarchy is created, with the grouping hierarchy informing the choices made in the reduction trees. Tenney and Polansky presented a formal model for melodic segmentation using the ideas of Gestalt psychology (1980). Lerdahl and Jackendoff also modeled their GPRs around Gestalt psychology. Others take a different approach. Melodic segmentation can assist the task of melodic reduction by identifying notes that are structurally more important. In all of the following research, a melody is defined as a monophonic sequence of notes, with no overlap. What this means is that no part of any two notes will sound at the same time. Melodic segmentation discovers pairs of boundaries that explicitly define subsequences within the sequence of notes. Certain melodic segmentation algorithms aim to apply this subsequence partitioning to both surface-level groups as well as higher-level groups. This section will document the research that has been performed for the purpose of melodic segmentation.

3.1.1. Implementing GTTM's Grouping Rules

There are some who chose to directly quantify the melodic segmentation work that Lerdahl and Jackendoff detailed in the GPRs of GTTM. As mentioned, the authors of GTTM did not provide a machine implementation of their work, however they aimed to specify the rules to the level of detail such that an implementation would be straightforward (Lerdahl and Jackendoff 1983, 53). Frankland and Cohen (2004) isolated and evaluated the part of the theory related to only to melodic segmentation, specifically the GPRs.

Upon reviewing the GPRs in GTTM, Frankland and Cohen found that only certain of the preference rules presented in GPRs 2 and 3 (discussed in Section 2.2.3) are translatable into mathematical equations. Specifically, the preferences for boundaries on larger differences in offset-to-onset interval (OOI) (GPR 2(a)), inter-onset interval (IOI) (GPR 2(b)), intervallic distance (GPR 3(a)), and duration (GPR 3(d)) were quantified (Frankland and Cohen 2004, 501).

A series of mathematical formulae were created to quantify GPRs 2 and 3 (see Table 3.1). Each formula was based on a particular parameter of the note, determined by the preference rule. The variable n represents the feature of the note (or pair of notes) that was being used, and is represented in the column labelled n. In the first row where the formula for GPR 2(a) is shown, the feature being utilized was the OOI between a set of two consecutive notes. The formula applied was simply the absolute length of the OOI, which can also be considered the amount of rest between the notes. The scale for the length of the rest was normalized so that a whole note, or *semibreve*, was valued at exactly 1.0. For GPR 2(b), IOI was used, which measures the timing difference between the attack point of two consecutive notes. The measurement involved a sequence of four notes, for which the three consecutive intervals between the four attack-points was measured. If the center interval was the largest in length, then the boundary strength was computed using the corresponding formula, otherwise a null value was assigned. The formula for GPR 3(a) measured the jump in consecutive pitch intervals. For a set of four notes, if the change in pitch interval between the middle two notes was the largest, then the boundary strength approached 1.0 as the middle pitch interval grew larger. For GPR 3(d), the duration of n_1 must be equal to the duration of n_2, and the duration of n_3 must be equal to the duration of n_4. Therefore, the formula simply compared n_1 and n_3. If they were equal, the value was zero, whereas if they were very different, the value approached 1.0. Given the mathematical formulae for the corresponding GPRs, the authors then compared the measurements with boundary judgements collected from participants listening to six different melodies. The only strong correlation with the empirical boundaries was produced with the boundary predictions from rule 2(b).

3.1.2. Local Boundary Detection Model

One computational approach to melodic segmentation is the Local Boundary Detection Model (LBDM) (Cambouropoulos 2001). The LBDM is similar to GTTM's GPR 3 in that it is designed to identify change in certain features of the consecutive intervals

GPR	Description	n	Boundary Strength
2(a)	Rest	OOI	absolute length of rest (semibreve = 1.0)
2(b)	Attack-point	IOI	$\begin{cases} 1.0 - \frac{n_1+n_3}{2\times n_2} & \text{if } n_2 > n_3 \wedge n_2 > n_1 \\ \bot & \text{otherwise} \end{cases}$
3(a)	Register Change	pitch	$\begin{cases} 1.0 - \frac{\lvert n_1-n_2\rvert+\lvert n_3-n_4\rvert}{2\times\lvert n_2-n_3\rvert} & \text{if } n_2 \neq n_3 \wedge \lvert n_2-n_3\rvert > \lvert n_1-n_2\rvert \wedge \lvert n_2-n_3\rvert > \lvert n_3-n_4\rvert \\ \bot & \text{otherwise} \end{cases}$
3(d)	Length Change	duration	$\begin{cases} \bot & \text{if } n_1 \neq n_2 \vee n_3 \neq n_4 \\ 1.0 - n_1/n_3 & \text{if } n_3 \geq n_1 \\ 1.0 - n_3/n_1 & \text{if } n_3 < n_1 \end{cases}$

Table 3.1.: The rules of GTTM's grouping theory, quantified by Frankland and Cohen (2004, 504–507), as gathered by Pearce etàl (2010, 6). The n column represents the feature being measured in the formula.

between notes. Each of these features were computed for every consecutive pair of notes in the input note sequence, creating a profile for each. The features included pitch interval in semitones, OOI and IOI. The *Change Rule* specifies a mathematical formula for the measurement of change over each of the interval profiles; a boundary strength s_i for a given interval x_i is a product of its surrounding intervals in the sequence:

$$s_i = x_i \times (r_{i-1,i} + r_{i,i+1})$$

Each r represents the degree of change between subsequent intervals:

$$r_{i,i+1} = \begin{cases} \frac{\lvert x_i - x_{i+1}\rvert}{x_i + x_{i+1}} & \text{iff } x_i + x_{i+1} \neq 0 \text{ and } x_i, x_{i+1} \geq 0 \\ 0 & \text{iff } x_i = x_{i+1} = 0 \end{cases}$$

Each of the features' strength profiles were then combined with a weighted average. The boundaries were found by identifying local peaks in the profile of the weighted average of strength profiles. An example is provided in Figure 3.1. Note that the change rule is based on the same principle as the GPRs in GTTM, by considering not only the current interval, but also the neighboring ones.

Cambouropoulos tested the weighted boundary strength profile with different weights for the different features, as well as a particular threshold over which a peak in the strength profile would be considered a boundary. The author chose a threshold such that 25% of notes fell on boundaries. The algorithm was evaluated on a data set of 52 Western tonal melodies with melodic punctuation notated by an expert performer. The term *melodic punctuation* was used to describe boundaries in melodic sub-phrases that are accentuated by a micropause in performance. These notations were created by previous research that was carried out to identify expressive timing deviations of melodic lines (Friberg, Bresin, Frydén, and Sundberg 1998). With the optimal parameters, LBDM was found to have a recall of .74 and a precision of .49 (Cambouropoulos 2001, 18).

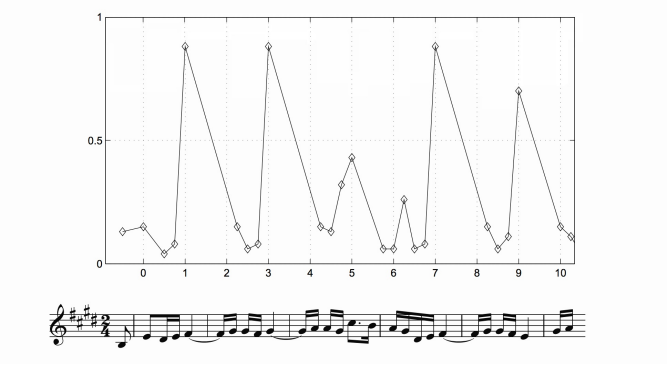


Figure 3.1.: An example of a boundary strength profile for Chopin's Etude Opus 10, Number 3 (Cambouropoulos 2001, 21). Each note's boundary strength is marked with a diamond in the graph.

3.1.3. Grouper

David Temperley (2001) also created a software system to automatically analyze music. His approach was strongly influenced by the ideas of GTTM—he even presented his rulesets using the terms: Well-formedness Rules and Preference Rules, borrowed from GTTM. Temperley utilized the Gestalt principles of proximity and similarity in an effort to discern what the listener experiences during a piece. In this work, Temperley presented six systems to analyze particular aspects of music: metrical structure, melodic phrase structure, contrapuntal structure, pitch spelling, harmonic structure, and key structure. Each of these systems provided their own set of well-formedness and preference rules, each of which defined the form of all possible structures, and the priority assigned between elements of the structure, respectively.

The software toolkit that provides the melodic phrase structure analysis is named Grouper (Temperley 2001), which consists of three main Phrase Structure Preference Rules (PSPRs). These PSPRs were paraphrased by Pearce, Müllensiefen, and Wiggins as follows (Pearce, Müllensiefen, and Wiggins 2010, 374):

> **PSPR 1 (Gap Rule):**
> Prefer to locate phrase boundaries at (a) large IOIs and (b) large OOIs; PSPR 1 is calculated as the sum of the IOI and OOI divided by the mean IOI of all previous notes;
>
> **PSPR 2 (Phrase Length Rule):**
> Prefer phrases with about 10 notes, achieved by penalising predicted phrases by $|(log_2 N) - log_2 10|$ where N is the number of notes in the predicted phrase—the preferred phrase length is chosen *ad hoc* (by Temperley (2001, 74)), to suit the corpus of music being studied (in this case Temperley's sample of the Essen Folksong Collection (EFSC) (Schaffrath 1995)) and therefore may not be general;

PSPR 3 (Metrical Parallelism Rule):
Prefer to begin successive groups at parallel points in the metrical hierarchy (e.g., both on the first beat of the bar).

3.1.4. Memory-Based Models

The previous models from Section 3.1.1 to 3.1.3 made boundary decisions either after analyzing an entire piece of music, or after making multiple parallel observations and combining them post hoc. There exists a class of models that utilize only a certain window of observations to predict the presence of a melodic segmentation boundary at a given point. These models are referred to as memory-based models, and include Markov models, n-grams, and probabilistic grammars.

Modelling Entropy

Ferrand, Nelson, and Wiggins (2003) applied a memory-based model to melodic segmentation. They formed their model on the assumption that "segmentation boundaries are likely to occur close to accentuated changes in entropy" (Ferrand, Nelson, and Wiggins 2003, 142). The authors defined entropy as the change of predictability associated with a certain sequence of events in a musical piece. Markov models with differing lengths of observational sequences were utilized to determine the probabilities for certain musical events. Often, the predictability of an event would change drastically at the beginning or end of a commonly occurring pattern. This technique can be performed on data that is not labelled with melodic segment boundaries, and was tested on Debussy's *Syrinx*, which contains segmentation data created by combining the boundary judgements from multiple listeners. The model was able to successfully predict all 11 boundaries, but also had 4 spurious predictions.

Data-Oriented Parsing Technique

One method utilizes a hierarchical model applied to musical melodic segmentation (Bod 2002). Bod borrowed from both machine learning and natural language processing in order to develop a grammar that determines phrase boundaries in melodies. To begin, the work analyzed and evaluated previous attempts at utilizing grammatical learning-based techniques, and also introduced a new method that incorporates memory, all evaluated on the EFSC (Schaffrath 1995). The algorithms were trained on 5,251 folksongs in the EFSC, and were tested on the remaining 1,000.

The work began by evaluating the Treebank grammar technique. The Treebank technique creates rules for all of the parse trees that literally occur in the training set. These parse trees are generated automatically, with one rule per phrase. For phrases, this means that each phrase, in its entirety, is a unique rule that specifies the entire note sequence of that phrase. Because of this, only the boundaries of phrases that have been seen before (or are identical to a seen phrase) can be identified. Unsurprisingly,

this method had a poor recall, at 3.4%, even though the precision was 68.7%; phrases that were repeated were easily interpreted and segmented, while completely new phrases were almost impossible to segment.

Bod compared his grammar-based methods with other works on the same dataset. Providing a more detailed view into phrases as well as how notes interact within a phrase is the Markov grammar technique (Seneff 1992; Collins 1999). It is similar to the Treebank method, except that every phrase is decomposed into a chain of Markov probabilities, with a certain history. For instance, if a six-note phrase is observed in the dataset, there will be a Markov chain of seven different probabilities, with all but the first conditioned on the previous n notes (n is determined by the history parameter). The seventh event is the token that represents the end of a phrase, which accounts for the additional probability computation. If the history parameter is set to four, and a phrase of length six is evaluated, the probability would be computed as follows (in this example P stands for a Phrase non-terminal, and *END* is a data point that demarcates end of a phrase). Each numeral digit is a single note event, represented by the index of its pitch in the current scale:

$$p(P \rightarrow 123456) = p(1) * p(2|1) * p(3|1,2) * p(4|1,2,3) * p(5|1,2,3,4)*$$

$$p(6|2,3,4,5) * p(\text{END}|3,4,5,6)$$

The probability for each note is conditioned on the history of the previous four notes, creating a fourth-order Markov chain on every observation, when available. Tested over the EFSC, this method had a slightly lower precision than the Treebank method (63.1%), however its recall was much improved (80.2%).

Bod presented his own grammar technique that is an enhancement of the Markov grammar. This technique corresponds to the Data-Oriented Parsing (DOP) technique developed earlier, also by Bod (1998). An inherent feature of this approach allows for the consideration of higher-level structure when determining phrase boundaries, as conditional probabilities. In this case, the number of phrases that occur in a folksong is the crucial information that allows for an improvement in segmentation performance. With the DOP, the conditional probability of a given phrase is based on the different higher-level rules in the grammar; for every folksong that is seen in training, a rule is created that specifies the number of phrases that was seen in that folksong. Here is an analogue to the Markov grammar example, using the DOP method:

$$p(P \rightarrow 123456|S \rightarrow PPP) = p(1|S \rightarrow PPP) * p(2|S \rightarrow PPP, 1)*$$

$$p(3|S \rightarrow PPP, 1, 2) * p(4|S \rightarrow PPP, 1, 2, 3)*$$

$$p(5|S \rightarrow PPP, 1, 2, 3, 4) * p(6|S \rightarrow PPP, 2, 3, 4, 5)*$$

$$p(\text{END}|S \rightarrow PPP, 3, 4, 5, 6)$$

In this case, the new rule of $S \rightarrow PPP$ stipulates that each melodic phrase should be part of a set of 3 phrases. The concept of applying more globally-defined features to the parsing of surface features was seen in the melodic segmentation models for

both the Grouper algorithm and the LBDM. Specifically, PSPR 2 in the Grouper algorithm (detailed in Section 3.1.3) also uses global information—it estimates the likely number of notes in each phrase, and uses that estimation to influence the decision of phrase boundaries (Temperley 2001). As in LBDM (described in Section 3.1.2) (Cambouropoulos 2001), the threshold for peaks in the strength profile to qualify as boundaries was based on the estimated percentage of notes that should be considered boundary notes. When Bod (2002) used this extra information in his model, the DOP parser obtained a precision of 76.6% and a recall of 85.9% on the EFSC.

The new design was able to find what the author calls "jump-phrase" boundaries—boundaries that occur between two consecutive notes of identical pitches, for which the prior note has a jump in pitch preceding it and the latter note has a jump in pitch succeeding it. Bod compared the boundary identification to hypothetical Gestalt boundary assignments, and asserted that neither Gestalt nor harmonic grouping preferences could explain this particular boundary. In the test set of the EFSC, over 32% of the folksongs had at least one of these jump-phrase boundaries, and overall, the total percentage of phrases that began or concluded with a jump was more than 15%.

IDyOM

Somewhat similar to the DOP method, another statistical method that instead employed unsupervised learning techniques was proposed by Pearce, Müllensiefen, and Wiggins (2010). The Information Dynamics Of Music (IDyOM) is used to model the information content, as defined by MacKay (2003), of a melody to determine which event in a sequence is more unexpected than the other events. Given a sequence of events (in this case, notes), and the preceding element, the model estimates the conditional probability of an event at index i with the formula:

$$p(e_i|e_i^{i-1}) \tag{3.1}$$

With the conditional probabilities, the measurement for the information content can be defined as such:

$$h(e_i|e_i^{i-1}) = log_2 \frac{1}{p(e_i|e_i^{i-1})} \tag{3.2}$$

h can be considered the measure of contextual unexpectedness of an event. Similarly the entropy of a given sequence is computed.

The premise presented in IDyOM is that melodic boundaries are found where the contextual unexpectedness and the entropy of a given note transition are high. The authors also extended their model to incorporate n-grams of a higher order by computing the frequency counts in the training stage for all n-grams up to a certain order. The authors then used a statistical method (Cleary, Teahun, and Witten 1995) for estimating probabilistic distributions using a weighted sum of all models in order to make their boundary predictions.

The authors of IDyOM also presented an evaluation of many of the aforementioned models, including Grouper, LBDM, and Frankland and Cohen's implementation of the GPRs. The dataset used for the evaluation was a subset of the EFSC (Schaffrath

Model	Precision	Recall	F1
Hybrid	0.87	0.56	0.66
Grouper	0.71	0.62	0.66
LBDM	0.70	0.60	0.63
IDyOM	0.76	0.50	0.58
GPR2a	0.99	0.45	0.58
GPR2b	0.47	0.42	0.39
GPR3a	0.29	0.46	0.35
GPR3d	0.66	0.22	0.31
Always	0.13	1.00	0.22
Never	0.00	0.00	0.00

Figure 3.2.: The results of different melodic segmentation systems (Pearce, Müllensiefen, and Wiggins 2010, 384).

1995), which contained 1705 Germanic folk melodies with corresponding annotations for melodic boundaries. They compared these models with IDyOM's performance, and also against two baseline models: one of which predicted that every event is a boundary, and another that predicted that no event is a boundary (labelled as "Always" and "Never" in Figure 3.2). Furthermore, they created a hybrid approach that included Grouper, LBDM, IDyOM, and the quantification of GPR 2a. This hybrid approach performed logistic regression on all of the models' predictions in order to combine them into one prediction for each event. The results for all methods are shown in Figure 3.2. For each method that was not from the authors of IDyOM, either the open-source framework was available (Grouper) or the method was specified to a level of detail necessary for implementation (LBDM, GPR 2a).

3.2. Melodic Parallelism

Melodic parallelism is a significant factor in the process of melodic segmentation, and, by extension, melodic reduction. However, there has been little research to formalize exactly how it influences the process. Parallelism can most simply be defined as the discovery of repeated patterns in music. Parallelism is a particular case of melodic similarity in that it aims to find significant repeated passages within the same piece. The term "significant" can have various definitions, which will be explored in the following sections.

Parallelism had an auxiliary focus in "The Perception of Musical Rhythm and Meter" (Steedman 1977), in which the author presented a process of discovering meter from symbolic music. Steedman determined higher levels of the metrical hierarchy by using repetitions of musical phrases of a certain durational value that appeared on the surface. For example, if a piece contains a musical phrase of three eighth notes that is immediately transposed and repeated, then the higher-level metrical unit will group the three notes

together into a single unit—the dotted quarter note. The group of 3 eighth notes can then be seen as durationally equivalent to other dotted quarter notes found in the piece. This often works well when considering the music of Bach, which the algorithm was designed to analyze, but can easily falter in other situations. The research by Steedman (1977) was built upon the author's earlier work with Longuet-Higgins (1971).

In (Ruwet 1972), the author used formal grammars as a verification tool for identifying segmentations in music. These segmentations were based on different types of musical repetitions—including both transformational repetitions, and oppositions or inversions.

Exact pattern matching is not a technique that is unique to music or melodic parallelism. Certain algorithms have solved the problem for efficiently discovering repeated patterns in strings (Crochemore 1981). This particular string-matching algorithm was repurposed by Cambouropoulos (2006) to find exactly-repeated phrases in a melody. In fact, depending on the representation of the melodic transitions, it is also possible to find repetitions of inexact pitch sequences. Cambouropoulos et al. (2005) accomplished this by creating interval categories that overlap, allowing the system to categorize a minor third as either a step or a leap. The exact same pattern extraction algorithm was used as part of a rhythmic pattern recognition model designed by Mont-Reynaud and Goldstein (1985).

As mentioned previously, the term "significant" in this context can describe certain characteristics of the patterns that are extracted with exact pattern-matching techniques. For example, Cambouropoulos (2006) argued that "significant" patterns can be quantified by a combination of length, frequency of occurrence, and degree of overlap. Each of these factors had a particular weight associated with it, and a *Selection Function* was created that maximized the selection of patterns that author considered best. Lartillot (2010) also created a system for prioritizing the different patterns that are extracted with pattern-matching techniques. Indeed, he utilizes some of the same features of patterns, such as length, frequency, and degree of overlap, but also adds the concept of cyclical parallelism—that is, those patterns that immediately repeat in a cyclical fashion. Furthermore, Lartillot applied a different concept for selecting the most "significant" patterns. First, the author defined a *maximal* pattern: a pattern that is not included in any other repeated patterns (Lartillot 2010, 203). As opposed to using a mathematical approximation, Lartillot borrowed the notion of *closed patterns* from information theory (Pasquier, Bastide, Taouil, and Lakhal 1999). A *closed pattern* is a pattern that is either *maximal* or occurs more frequently than the pattern of which it is a part. An example of a score analysis for discovering parallelism is shown in Figure 3.3. In this example, the *maximal* pattern is simply pattern P, because it subsumes all other repeated patterns. However, as Lartillot pointed out (2010, 205), if the selection method is *maximality*, then patterns a and ab would be discarded, since only one pattern can be *maximal*. This would be detrimental because the most frequently repeated pattern is, in fact, a. By using the concept of *closed patterns* as the selection method, more of the relevant patterns are kept. In this case, those patterns that are more frequent than their parent pattern are pattern ab, which occurs four times compared to pattern P, which occurs twice, and pattern a, which occurs six times.

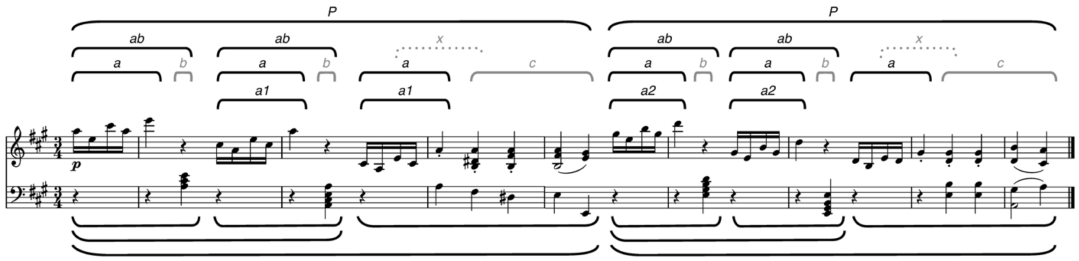


Figure 3.3.: An example analysis of the repeated patterns found in Beethoven's Sonata, Opus 2, Number 2, using the computational framework built by Lartillot (2010, 202). The *closed* patterns are represented by the a, ab, and P patterns, while b, c, and x are *non-closed* because they are either less frequent than or equally frequent to their parent patterns.

3.3. Melodic Reduction

While the task of melodic reduction often requires the analysis of melodies for auxiliary purposes (like those in Section 3.1 and Section 3.2), certain methods do attempt to mechanize melodic reduction holistically. Some of these methods create piece-wise systems that address multiple facets of the reduction challenge and combine them into a single solution, while others are designed in such a way that they can address the problem directly.

3.3.1. Musical Grammars and Trees

When grammars and trees are applied to music analysis, they intrinsically reduce the musical input because of their hierarchical nature. Many have exploited this feature in order to identify similar melodies, or variations, or to generate new melodies.

In 1979, utilizing grammars for music was already of much interest, such that a survey of the different approaches was in order (Roads and Wieneke 1979). Ruwet (1975) suggested that a generative grammar would be an excellent model for the creation of a top-down theory of music. Smoliar (1976) attempted to decompose musical structure (including melodies) from audio signals with a grammar-based system. Kassler (1963) suggested that a generative grammar would also fit the form of the theory of musical diminution presented by Heinrich Schenker .

Baroni et al. (1982) also created a grammatical system for analyzing and generating melodies in the style of Lutheran chorales and French chansons. The computer program would create a completed, embellished melody from an input that consisted of a so-called "primitive phrase" (Baroni et al. 1982, 208). A "primitive phrase" consisted of a melodic backbone that lays the foundation for the creation of additional surface notes. However,

it seemed the problem of composing a melody with more long-term form was prevalent, as Smoliar articulated in his review of the literature (Smoliar 1986, 137):

> [T]he programs which are discussed are similar to a program which can generate sentences from a given grammar. The resulting sentences are grammatical but generally weak in meaning. Such a program has yet to produce a set of sentences which coherently form a paragraph.

Gilbert and Conklin (2007) designed a grammar for melodic reduction based on the Repeat Rule, Neighor Tone Rule, Passing Tone Rule, and the Escape Tone Rule as described in Chapter 4, and utilized unsupervised learning on 185 of Bach's chorales from the EFSC. This grammar was also utilized by Abdallah and Gold (2014), who implemented a system in the logical probabilistic grammar framework PRISM for the comparison of probabilistic systems applied to automatic melodic analysis. The authors implemented the melodic reduction grammar provided by Gilbert and Conklin using two separate parameterizations and compared the results against four different variations of Markov models. The evaluation method was based on data compression, so that those models that performed better were those that required the least amount of information to encode a given note. The evaluation metric was given in bits per note (bpn). Tested over four separate subsets of the Essen Folksong Collection, the authors found that the grammar designed by Gilbert and Conklin was the best performer with 2.68 bpn over all the datasets, but one of the Markov model methods had a very similar performance. The same authors also collaborated with Marsden (2016) to detail an overview of probabilistic systems used for the analysis of symbolic music, including melodies.

Other work sought to implement the generative grammar that was presented by Lerdahl and Jackendoff in GTTM. Hamanaka et al. (2007b) presented a system for implementing GTTM. While the original theory was intended to be a grammar, Hamanaka et al. utilized many mathematical models for estimating preferences and priorities so that the rules of the grammar actually manifested in a system of probabilities and constraints. The resulting framework identifies time-span trees automatically from monophonic melodic input. Time-span trees were chosen for implementation rather than prolongational trees because the time-span trees relate more directly to the grouping structure analysis and the metrical structure analysis rules also presented in GTTM. Some of the preference rules for grouping structure and metrical structure are more straightforward to quantify. Furthermore, the time-span trees do not consider harmony, while the prolongational trees do. The authors parameterized 17 of the 22 rules that are formalized in GTTM. For each rule, Hamanaka et al. assigned a strength in order to quantify priority between the rules. Parameters were also created for particular rules that require them. For example, a parameter is necessary for indicating how important the length of a phrase is when considering parallel phrases. In order to utilize this rule for creating boundaries, one must consider whether or not parallel passages should be notated with a boundary at either the beginning or the end of the parallel phrase. A parameter was also created to indicate where the boundary should be placed.

Each of the parameters were tuned by hand on a piece-by-piece basis. The authors contended that the parameters likely reflect stylistic elements of a particular piece, or

family of pieces (Hamanaka, Hirata, and Tojo 2007b, 276). A dataset of 100 solutions was created specifically for testing the efficacy of the time-span tree analyzer. Expert musicologists were instructed to follow GTTM and apply all of its rules in order to create these time-span tree solutions. Using the hand-tuned parameters and testing against the dataset of solution trees, the system attained an f-measure of 0.60.

Later, Hamanaka et al. (2007a) created a system that would automatically estimate all the parameters, obtaining an f-measure of 0.35—still above the baseline f-measure of untrained parameters.

Hamanaka et al. were not the only ones to endeavor to make systems that implemented GTTM for the purposes of melodic reduction. Nord (1992) created a system to prove the theoretical value of GTTM, and applied all the rules by hand. Others used the theories within GTTM to create compositional frameworks or aids, as will be discussed in Section 3.4.

Bernabeu et al. (2011) created a Probabilistic Context-Free Grammar (PCFG) for reducing melody that was used to find inexact matches of similar melodies. The reductive properties of the trees allowed for comparisons with melodies that had similar, but not exact pitch sequences. The algorithm was trained on 420 monophonic 8–12 bar incipits of 20 universally-known melodies. The test set was a MIDI representation of the same melodies performed by both amateur and professional pianists. Each incipit was performed by multiple pianists, giving multiple samples of the same melody with slight differences in timing and pitch due to performance errors. The resulting performance was a 87.3% success rate.

Kirlin and Jensen (2011) and Kirlin (2014) modelled the hierarchies in melodies using probabilistic methods and triangular graphs, which have a similar form to a single grammatical rule with two right-hand side values.

3.4. Automatic and assisted composition

There are times that melodic reduction is performed for the purpose of reversing the process to create new melodies or melodic fragments. In other research, structures that are similar to those used in melodic reduction tools (such as tree structures) are also used for compositional tools. This section will explore compositional tools created using melodic reduction and tree structures.

Lerdahl built upon his own work by implementing a system for assisted composition (Lerdahl and Potard 1986). In this work, the authors utilized both the grouping structure and the prolongational reduction frameworks of GTTM. The motivation for using those particular theories was that the grouping structure would give a sense of continuity and temporal locale, while the prolongational structure would ensure that tension and resolution is innate to the system (Lerdahl and Potard 1986, 17). From the prolongational reduction theory, they borrowed the notation and terminology for strong prolongations, weak prolongations, and progressions. Each branch of the prolongational tree thus is referential to the corresponding parent event. Lerdahl and Potard were focused on the creation of a software tool that would allow composers to

sketch out different aspects of a piece before knowing exactly what notes they wanted to score. Therefore, the tool allowed the user to either define the relative prolongational movements, and assigned the metrical structure based on that, or the user could do the opposite—define the metrical structure of a piece, and have the software assign the prolongational form. In this way, authors were seeking to create a musical sketch pad that could interface with the psychological motivations of the composer, no matter what part of the musical piece the composer had formed.

The Musical Object Development Environment (MODE) framework was also built loosely upon the prolongational tree structures (Pope 1991b). Pope preferred to refer to the trees as T-R trees, for Tension-Relaxation. MODE is a framework with a graphical user interface (GUI) for the editing and application of T-R trees for the purpose of musical composition. Figure 3.4 shows an example of the GUI used to manipulate Chinese poetry when set to music.

GTTM was not the only theory that utilized a formal grammar for the goal of automatic composition. Roads explored the concept of "Composing Grammars" (Roads 1977). In this work, he created a compositional tool called COTREE that used context-free rewriting rules in order to help a composer generate music.

Cope created a system for generating music in which similar musical phrases were automatically created using a linguistic technique called the Augmented Transition Network (Cope 1996).

Baroni and Jacoboni designed a grammar that was utilized to analyze and generate melodies in the style of major-mode chorales by Bach (Baroni and Jacobini 1975; Baroni and Jacoboni 1978). The output of the system would generate the soprano part of the first two phrases of the chorale.

The Bol processor was implemented as a system that would generate tabla phrases using a grammatical representation of tabla performance (Bel and Kippen 1992). The system analyzes the performance of tabla phrases by expert players, and then constructs new phrases from the resulting grammar.

The hierarchical nature of a score was also exploited by means of a grammar by Buxton, et al. (1978). In this work, the authors created a recursive object in order to represent a part of the score. This object could represent an event as small as a single note, or a set of events that spanned the entire score. This musical event object was specified with a grammar, and could recursively expand to accommodate the different sizes. Buxton et al. could then, for example, specify a grammar tree that represented a motif, and use it as a template for theme and variation. The template would be applied repeatedly in succession, and elaborated in a different way using the same grammar for each iteration.

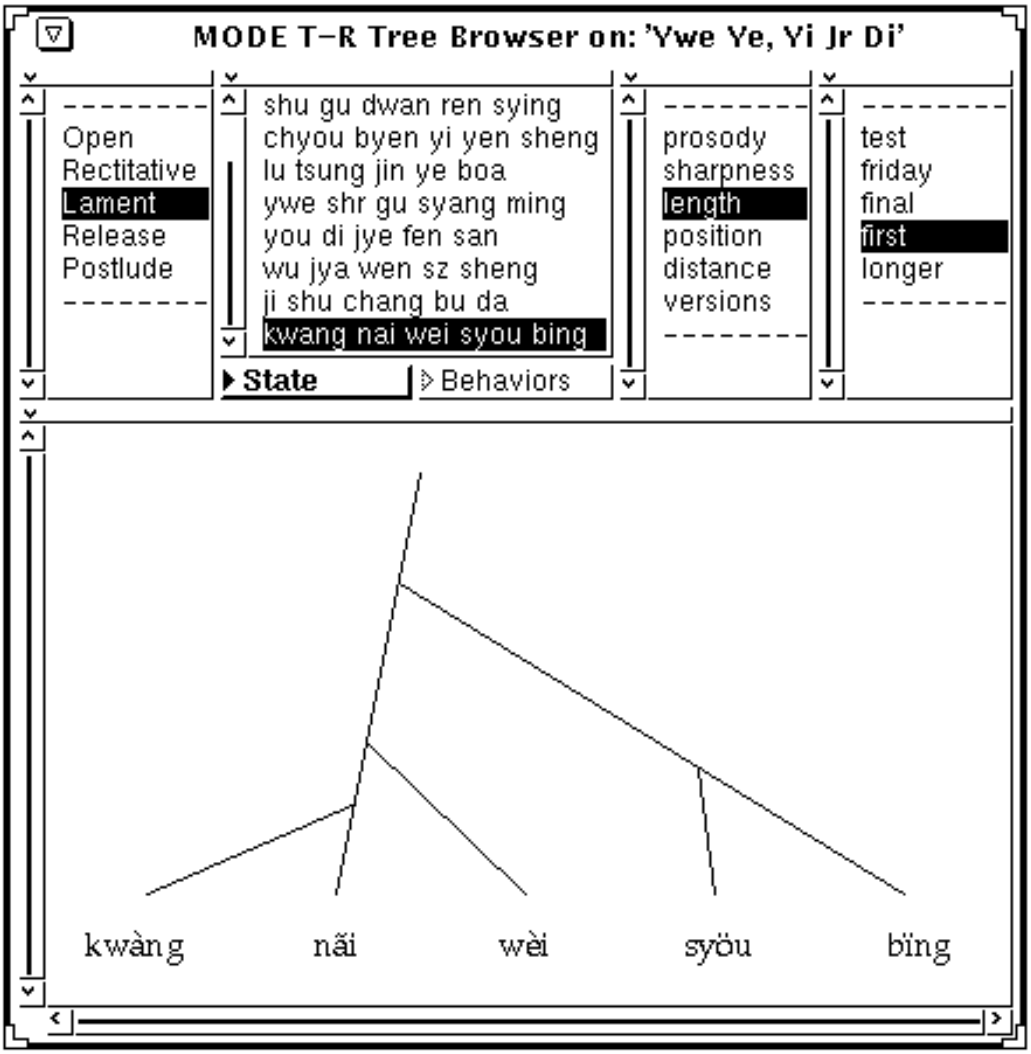


Figure 3.4.: A Tension-Relaxation (T-R) tree example shown in the MODE GUI provided by Pope (1991a, 326). This example shows the text from a Chinese poem entitled *Moonlight Night* being set to music, with the durational T-R tree visible (note the parameter "length" is highlighted in the third column at the top). Manipulating the locations of the leaf words in the tree would affect their duration when synthesized along with the music.

4. A PCFG for Melodic Reduction

The focus of this research is to encode the rules of melodic embellishment into a Probabilistic Context-Free Grammar (PCFG). Each embellishment rule can be transferred directly to either one or multiple probabilistic production rules. Using these rules and a dataset of hand-annotated tree solutions that represent reduced notes, one can train the PCFG, and apply the PCFG to new test cases. The trained PCFG will be tested for accuracy against solutions outside of the training set. As a starting point, the Context-Free Grammar (CFG) that will represent the grammatical melodic reduction framework must be formed.

The rules of melodic embellishment are so widely used that they are detailed in a musical dictionary (Sadie and Grove 1980). In order to facilitate the understanding of the embellishment rules, some definitions will be of use. In Western tonal music, pieces generally contain a restricted subset of notes that have a particular relationship to each other. These relationships are called musical scales. The fundamental unit that measures pitch distance in a scale is the semitone. Generally, each step in a scale will represent one or two (and sometimes three) semitones. The relationship between this ordered set of notes is distinct in the intervals between each step. In the case of major scales that is W-W-H-W-W-W-H, where 'W' represents a whole tone, or two semitones, and the 'H' represents a half tone, or semitone. When a scale with this order of relationships is used as the fundamental organization of a piece, it is referred to as the *diatonic* framework. Within the diatonic framework, a step is the difference between two consecutive notes of a scale. This is an important consideration because many embellishment rules are used in the context of the diatonic framework, and thus the intervals between notes can be either one or two semitones.

An embellishment is considered any note that can be added to another more structural note to create more variety in the piece. In the course of this research, the structural importance of a note is directly related to its place in the tree hierarchies provided in the input data. The embellishment rules considered for building the CFG have a couple of specific features. It was desirable to exclude any contextual information in the rules—like harmony—so rules that require harmony were eliminated for consideration. Furthermore, because the rules will need to rely on their form as opposed to their context, it would be beneficial to use rules that have a specific form. For this reason, rules that span three notes, as opposed to only two, were selected. The rules used in the course of this experiment were as follows:

- **Neighbor Tone Rule**

  For any given note, it is possible to embellish the note by adding a new note just one diatonic step away in pitch, and returning afterwards to the original note.

- **Escape Tone Rule**

  In the situation in which there are two consecutive notes that are separated by a leap (more than one diatonic step), one can embellish the first note by moving by one diatonic step in the opposite direction of the leap. The escape tone is also known as the échappée.

- **Cambiata**

  The musical situation defined by the Cambiata is a leap followed by a step in the opposite direction. It is similar to an Escape Tone, however the order of the interval sizes are reversed. Since it has such a specific shape, it will also be included.

- **Passing Tone Rule**

  When two consecutive notes form a leap with a magnitude of two diatonic steps, it is possible to insert a note in between them that is one diatonic step above the first note and one diatonic step below the second. This forms an ascending passing tone. Descending passing tones are also possible.

- **Repeat Rule**

  One simple way to embellish a note is to simply break up the duration of the note into two separate notes of the same pitch.

It is the basis of these five rules that we will form the CFG. This set of rules is almost the same as the ruleset presented in previous work with a PCFG for melodic reduction (Gilbert and Conklin 2007). The authors used a ruleset consisting of the Neighbor Tone, Escape Tone, Escape Tone, Passing Tone, and Repeat Rule, excluding the Cambiata Rule. A visualization of their ruleset can be seen in Figure 4.1. Conklin and Gilbert also added an additional rule that accounted for any interval that could not be described by one of the embellishment rules. There are also rules for embellishment that won't be incorporated:

- **Suspension**

  A melodic suspension occurs when there is change of the underlying harmony, and the resolution to a new note that is part of the underlying harmony is delayed; instead there is a note before the resolution that is one diatonic step above the eventual resolved note. Thus, the final note is *suspended* in both its onset timing and by the higher pitch of the preceding note.

- **Anticipation**

  Anticipation occurs also on a harmonic change. One of the notes that is part of the new harmony is played just before the harmonic change, and then again when the change happens. Often the note is dissonant with the harmony when it is played before the change, so it provides a tension before resolving on the same note with a consonant underlying harmony.

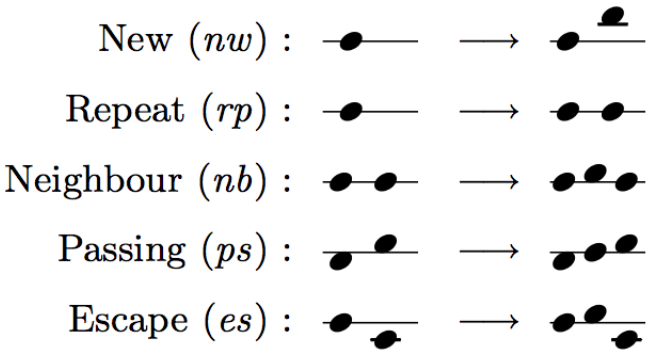


Figure 4.1.: A visualization of a set of melodic embellishment rules, encoded manually into the production rules of a formal grammar (Gilbert and Conklin 2007, 3).

- **Consonant Skip**

  A consonant skip is simply a leap from one note in the harmony to another note in the harmony.

The embellishments immediately above represent the rules that do not span exactly three notes in length, require harmonic context, or both. They were excluded from the CFG.

4.1. Approach

The method of analysis presented in this work will involve the assessment of a PCFG for melodic reduction by the means of supervised training and cross-fold validation. The process will occur as follows:

1. Construct the CFG from the selected embellishment rules.
2. Convert the database of tree solutions into the tree format that is usable by a CFG.
3. Use supervised training to calculate the probabilities of each rule in the CFG, based on the converted solution trees.
4. Run the trained grammar on the test melodies, creating a tree parse for each test based on the most probable rules from the training stage.
5. Compare the tree parses generated from the trained grammar with the tree solutions from the test set.

With cross-fold validation (Jurafsky and Martin 2000, 154), the data is segmented into a number of segments (called "folds"), and for each iteration one segment is held as the test set, and the other segments are used as the training set. The training and testing steps are then performed, and a new segment is chosen as the test set. The results are aggregated across all folds, in order to average out any inconsistencies that might be created by selecting a particular configuration.

Using this approach, it will be possible to determine whether the time-span trees or the prolongational reduction trees can be used to create an effective tool for automatically reducing melodies in the framework of a PCFG.

4.2. Creating the CFG

In order to create a PCFG, a CFG must first be designed. While the rules are described in diatonic terms, the CFG will be implemented without the context of a musical key. The CFG will compare pitch intervals between pairs of consecutive notes in a score, measured by an integer number of semitones. Say, for example, if a CFG contains an implementation for the Neighbor Tone Rule, its rules would need to cover all of the semitone intervals that represent a diatonic step. Namely, 1 and 2. The rules would naturally follow:

$$
\begin{aligned}
0 &\rightarrow 1\ \text{-}1 \\
0 &\rightarrow 2\ \text{-}2 \\
0 &\rightarrow \text{-}1\ 1 \\
0 &\rightarrow \text{-}2\ 2
\end{aligned}
$$

This ruleset is incomplete because a CFG was designed for string-based representations, so the semitone intervals on the right-hand side must also be resolved with their string counterpart. For each semitone interval that occurs on the right-hand side of a rule, then, we need an additional rule to expand that interval into its string-based representation:

$$
\begin{aligned}
0 &\rightarrow 1\ \text{-}1 \\
0 &\rightarrow 2\ \text{-}2 \\
0 &\rightarrow \text{-}1\ 1 \\
0 &\rightarrow \text{-}2\ 2 \\
1 &\rightarrow \text{`1'} \\
\text{-}1 &\rightarrow \text{`-1'} \\
2 &\rightarrow \text{`2'} \\
\text{-}2 &\rightarrow \text{`-2'}
\end{aligned}
$$

Now we have a CFG that can processing a string-based input, as it was originally designed to do. There is one further issue with the implementation; the left-hand side non-terminals follow a more strict naming convention that excludes the use of the minus sign as the first character. Thus the CFG need be modified further to prevent this from happening. For each negative non-terminal, the minus sign is replaced with the letter 'm':

$$
\begin{aligned}
0 &\rightarrow 1\ \text{m1} \\
0 &\rightarrow 2\ \text{m2} \\
0 &\rightarrow \text{m1}\ 1 \\
0 &\rightarrow \text{m2}\ 2 \\
1 &\rightarrow \text{`1'} \\
\text{m1} &\rightarrow \text{`-1'} \\
2 &\rightarrow \text{`2'} \\
\text{m2} &\rightarrow \text{`-2'}
\end{aligned}
$$

For the Neighbor Tone Rule, as seen above, the grammar only needs to expand a small subset of intervals to cover all the possible diatonic steps and to keep the relationship of the intervals intact. For other rules this process is slightly more complicated. Specifically, for the Escape Tone Rule as defined above, the second interval in the right-hand side of the rule specifies that there must be a leap (more than one diatonic step). There must be a limit as well on the maximum number of semitones, otherwise the rule would infinitely expand. Also, it wouldn't make sense to have semitone intervals outside of a certain range, because the human ear can only hear pitches of a particular range, and thus those pitches outside that range are not used in music. For purposes of the Escape Tone Rule and its corresponding leap, we apply a limit of twenty-four semitones, which equates to a two-octave leap. We'll know if this limit is inadequate if there exists any interval in the corpus that extends beyond twenty-four semitones. With this limit in mind, we can expand the Escape Tone Rule also into a CFG format:

$$
\begin{aligned}
\text{m23} &\rightarrow 1\ \text{m24} \\
\text{m22} &\rightarrow 2\ \text{m24} \\
\text{m22} &\rightarrow 1\ \text{m23} \\
&\vdots \\
\text{m2} &\rightarrow 2\ \text{m4} \\
\text{m2} &\rightarrow 1\ \text{m3} \\
\text{m1} &\rightarrow 2\ \text{m3} \\
1 &\rightarrow \text{m2}\ 3 \\
2 &\rightarrow \text{m1}\ 3 \\
2 &\rightarrow \text{m2}\ 4 \\
3 &\rightarrow \text{m1}\ 4 \\
3 &\rightarrow \text{m2}\ 5 \\
&\vdots \\
22 &\rightarrow \text{m1}\ 23 \\
22 &\rightarrow \text{m2}\ 24 \\
23 &\rightarrow \text{m1}\ 24
\end{aligned}
$$

For this embellishment rule, the rules that would incorporate right-hand side values of more than twenty-four are excluded. Specifically, the production rules with a left-hand side of the non-terminal '23' only amount to a single rule, as opposed to two rules, which is the normal set of combinations for each interval. The reason for this is that it would necessitate an interval of twenty-five on the right-hand side of the rule. This is already a

limitation, but again should not be a problem is that particular situation does not occur in the training and test sets.

One other thing of note is the expansion of the CFG rules that are required to describe even the limited ranges of the left-hand side values. Here, one left-hand side value is a range of [-24:-3] and [3:24], while the other is [1:-2], and [-1:-2]. We end up with 42×4 possibilities, leaving us with one hundred and sixty-eight productions. The number of productions involved with a CFG directly affects the ability to parse input sequences. If the number of productions grows to be too large, the parsing problem could become intractable. One benefit of the PCFG is that the rules will only be incorporated into the trained PCFG if they have occurred in the training set. So there is potential that the training process will prune zero-probability rules in the CFG.

Besides the rules specifically designed for musical rules of embellishment, one extra rule is necessary. The embellishment rules will not exhaust all possible combinations of intervals; there will be certain interval sequences that satisfy none of the encoded embellishment rules. Therefore, a rule must be created that allows any new interval to happen, so that the grammar can describe every possible melodic sequence. The rule created describes any interval that does not qualify as a melodic embellishment. First, we must define "any interval", by specifying a rule that has every possible interval in a series of right-hand side extrapolations. The token chosen to represent this "any interval" is the letter 'N', for "New". Using the previously-specified range of intervals from [-24:24], the "New" rule will expand each possibility:

$$
\begin{gathered}
\mathrm{N} \rightarrow \mathrm{m24} \\
\mathrm{N} \rightarrow \mathrm{m23} \\
\mathrm{N} \rightarrow \mathrm{m22} \\
\vdots \\
\mathrm{N} \rightarrow \mathrm{m1} \\
\mathrm{N} \rightarrow 0 \\
\mathrm{N} \rightarrow 1 \\
\vdots \\
\mathrm{N} \rightarrow 22 \\
\mathrm{N} \rightarrow 23 \\
\mathrm{N} \rightarrow 24
\end{gathered}
$$

In order to properly utilize the "New" rule, it must optionally be recursive, so that any number of new intervals can be found in sequence. The following rule describes exactly that:

$$\mathrm{N} \rightarrow \mathrm{N}\ \mathrm{N}$$

Lastly, as with every CFG, a starting node must be specified. Conventionally an 'S' is chosen to denote the starting rule, which will simply expand to a "New" rule:

$$\mathrm{S} \rightarrow \mathrm{N}$$

In order to generate all the rules for the CFG, a system was built that specifies the constraints for each item in the production. The corresponding CFG rules are then generated in string format, as depicted above. Using this system, it is easy to generate new constraint-based rules and automatically create the corresponding CFG. The details of the implementation for this system will be specified in section 5.1.

4.2.1. Alternative Representations

It is possible to reduce the number of rules by changing the representation of the data. If the basic unit of data was a diatonic interval instead of a semitone interval, for example, there would be only seven possibilities per octave as opposed to twelve. This would also have the benefit of grouping the production rules for both the Neighbor Tone Rule and the Passing Tone Rule. In the case of the Neighbor Tone Rule, it would require only two production rules for the embellishment rule, as opposed to the four rules shown above. The two rules would be:

$$0 \rightarrow 1 \text{ m1}$$
$$0 \rightarrow \text{m1 } 1$$

This would similarly reduce the ruleset for the Passing Tone Rule. However, there would be an up-front cost of converting the data into its corresponding diatonic indices, using the key information.

Another possibility is to limit the semitone intervals to their octave-equivalent pitch class. With this method, an interval of fourteen semitones would be represented as simply an interval of two semitones. This representation would also collapse the number of rules required, since any semitone interval greater than eleven would be removed from the dataset. This has one unfortunate side-effect in that, for example, a leap of thirteen semitones would no longer be considered a leap, because it would be represented as an interval of one.

Perhaps a level of abstraction would be of greater use, to group the embellishment rules more succinctly. It is also possible to represent the data as either a leap or a step, since the embellishment rules selected rely entirely on that distinction. The unfortunate drawback is that the precise interval is then lost. If the goal were to sample from the PCFG in order to create new melodies from the trained model, then this level of abstraction would prevent precise intervals from being generated.

4.3. Grammar Induction and Parsing

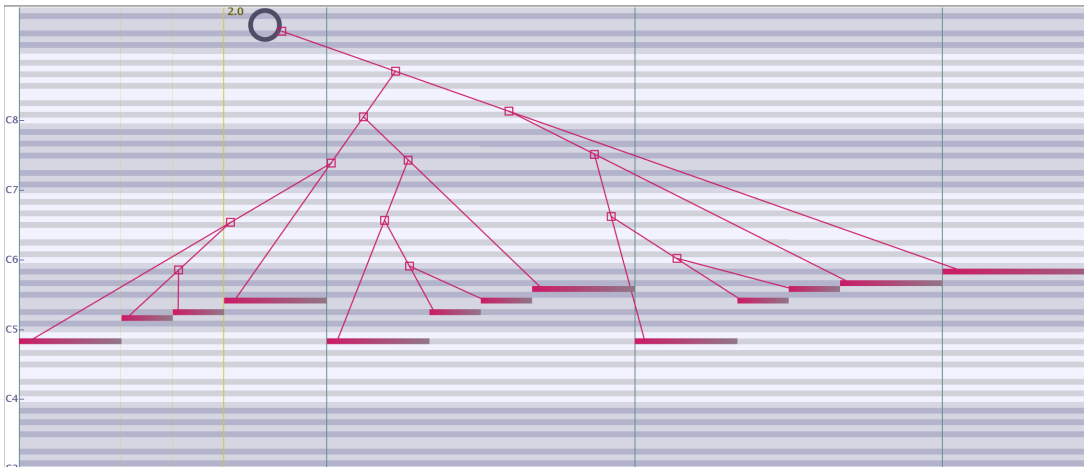
In order to convert the CFG into a PCFG, data is required. The dataset that is used for training the PCFG is the hand-made melodic analyses using The Generative Theory of Tonal Music (GTTM) provided by Hamanaka, Hirata, and Tojo (2007b). This dataset contains three hundred separate analyses of melodies using GTTM. The analyses are split into two groups that vary only in the number of analyses each group provides. Specifically, for all three hundred melodies, there are three separate expert analyses

that represent the different features of the melodies that GTTM addresses: Grouping Preference Rules (GPRs), Metrical Preference Rules (MPRs), and the Time-Span Preference Rules (TSPRs). Additionally, for a smaller set of one hundred melodies, the Prolongational Reduction Preference Rules (PRPRs) are applied. The TSPRs and the PRPRs result in tree structures (time-span trees and prolongational reduction trees), while the GPRs are represented as subsequences of notes that are perceptually grouped, and the MPRs are represented with a hierarchical dot-based structure. All formats are encoded in the Extensible Markup Language (XML) file format. Only the TSPRs and the PRPRs can be used as input for supervised learning of the PCFG, because a grammar requires a tree-based structure as input for the learning process. The main difference between the PRPRs and the TSPRs is that the PRPR analyses include decisions made with the influence of harmonic context. Because this research has excluded rules that require a harmonic context, it would make sense to utilize the trees created from the TSPR rules (time-span trees) instead. Additionally, the time-span tree analyses amount to three hundred analyses, while the prolongational trees only amount to one hundred in this particular dataset. With supervised learning of probabilistic systems, it is better to have more data so that there are less chances of seeing a particular observation that has a zero probability. This research will run both datasets separately in order to determine which is a better fit for the experiment.

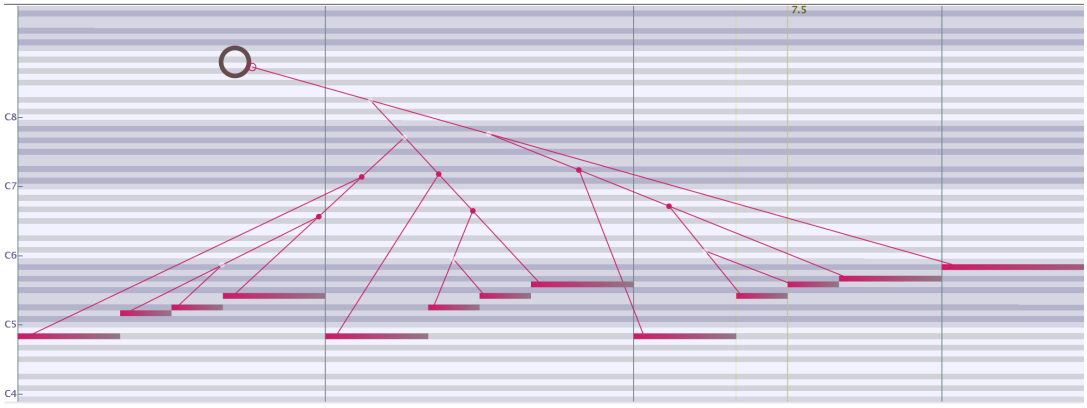
4.3.1. Formatting the Training Data

There is an additional step in order to use the GTTM dataset for supervised training of the PCFG. The input format provided by GTTM is slightly different than what is required to train the PCFG. As was discussed in Section 2.2.4, the time-span trees differ from grammar trees in that one element will *subsume* another, creating a direct and unique parent-child relationship. With the grammatical rules that were selected for the PCFG, each note is subordinate to not just one note, but two separate notes. Because of this, the time-span trees must be converted to the grammar-based format in order to inform the PCFG of the solutions. Take for example the first melody in the GTTM dataset, Frédéric Chopin's "Grande Valse Brillante", a waltz in E flat. Shown are both the time-span tree (in Figure 4.2a) as well as the prolongational reduction tree (in Figure 4.2b).

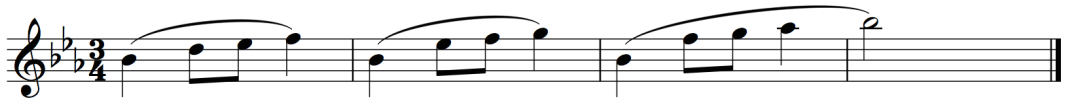
The GTTM trees are encoded in XML. Each XML file annotates a tree in which each level consists of both a `primary` and `secondary` tag, each of which can optionally be recursive. Every tag will at a minimum contain a `head` node, which specifies the note in the original score that it references, with a format of `<Part Name>-<Measure Number>-<Note Number>`. An example melodic phrase represented in the XML format of the GTTM dataset is given in Appendix B. The XML listed there represents the time-span tree of the second group of four notes in the melody of "Grande Valse Brillante". Both the time-span tree and the prolongational tree for the same four notes are displayed in Figure 4.3. This phrase is a prime example of the differences between the time-span and prolongational reductions. Specifically, the time-span tree attaches the middle two notes to the left-most branch. Intuitively this makes sense



(a) The time-span tree for half of the first melody in the GTTM dataset, Frédéric Chopin's "Grande Valse Brillante", as displayed in the GTTM visualizer provided by Hamanaka, Hirata, and Tojo (2007b).



(b) The prolongational reduction tree for half of the first melody in the GTTM dataset, Frédéric Chopin's "Grande Valse Brillante", as displayed in the GTTM visualizer provided by Hamanaka, Hirata, and Tojo (2007b).



(c) Half of the melody, in musical score format, for Frédéric Chopin's "Grande Valse Brillante".

Figure 4.2.: An example score and the corresponding time-span tree and prolongational reduction tree.

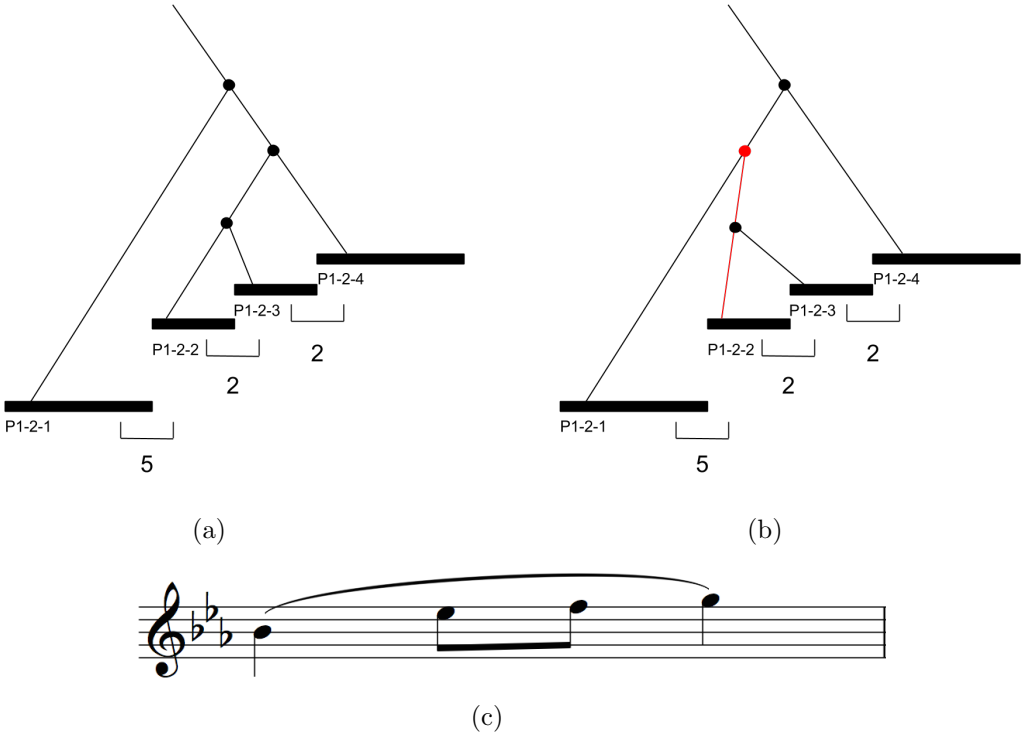


Figure 4.3.: The prolongational tree (a) and the time-span tree (b) for the second four notes in Frédéric Chopin's "Grande Valse Brillante", as well as the corresponding score (c). The notes are labelled as in the XML, and the intervals between them are notated in number of semitones. The time-span tree contains a problematic branch that prioritizes metric level over pitch relationships.

because the time-span trees are informed by the metrical hierarchy, and the left-most note represents the downbeat of the measure, and thus is a more prominent rhythmic onset. The prolongational reduction, on the other hand, represents pitch relationships based on harmony, and therefore the the second note in the phrase branches off of the final note of the phrase. Both of the notes are part of the underlying harmony in that measure, and with the passing tone in between them they represent a tension created by moving away from a harmonic tone and a subsequent relaxation when returning to a harmonic tone. The prolongational trees aim to notate these tension and relaxation relationships.

The CFG tree format is slightly different from the form found in the GTTM dataset. With the CFG, most of the embellished notes require two parents that surround the original note temporally—that is, the embellished note must follow the left parent and occur before the right parent, in the input sequence. With three notes, two intervals can be created, which is the form of all the embellishment rules in the CFG.

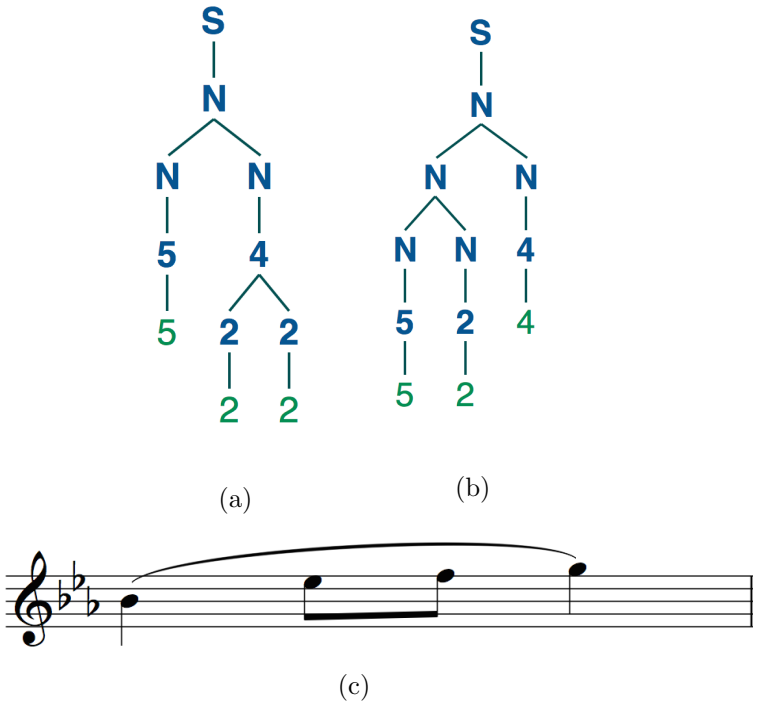


Figure 4.4.: The converted trees from the prolongational tree solution (a) and the time-span tree solution (b) from the GTTM dataset for the second set of four notes in Frédéric Chopin's "Grande Valse Brillante", as well as the corresponding score (c). The intervallic data is lost for the time-span tree parse when the `N` $\rightarrow$ `N N` rule is applied.

The Algorithm

An algorithm was created to recursively iterate through all the branches in a particular GTTM tree, and to apply the grammar at each branch configuration. The algorithm will iterate from the bottom up, selecting first the `primary` and `secondary` head notes, and then returning their information to the higher level. Then, the `primary` and `secondary` notes will be combined with the closest note from the parent tag. So, in the prolongational tree example from Figure 4.3a, note `P1-2-3` will be the note of least importance, and will be paired with note `P1-2-2`. `P1-2-3` is the `secondary` tag and `P1-2-2` is the `primary` tag. Since, at this level, there are only two possible notes, and the CFG rules require three, this pair is returned to the next highest level to complete the triple. When the pair is tupled with `P1-2-4`, then the algorithm will compute the intervals between the three notes, and apply the grammar. This set of three notes represents the Passing Tone Rule perfectly, and thus note `P1-2-3` will be reduced out. The CFG rule that would be used is the `4` $\rightarrow$ `2 2` rule, so that the resulting interval would be four semitones. This process continues recursively up the tree until the CFG has been applied to all branch combinations from the original tree. In the case of this particular example, the CFG will next be applied to notes `P1-2-1`, `P1-2-2`, and `P1-2-4`. For this set of notes, there is no embellishment rule that describes the two consecutive jumps of an interval of five semitones followed by an interval of four semitones, so the "New" rule will be applied, and the process will conclude. The resulting parse tree that is produced is represented in Figure 4.4a.

One challenge is that the note of the lowest priority in a group of three notes may not be the note that is found in the temporal center of the three notes in the score. This happens in the time-span tree in Figure 4.3b, between notes `P1-2-1`, `P1-2-2`, and `P1-2-3`. In this case, both branches are of the same direction—to the right. The note of lowest priority is `P1-2-3`. Ideally, the CFG should be applied to notes `P1-2-2`, `P1-2-3`, and `P1-2-4` in order to reduce `P1-2-3`, but that would violate the hierarchy specified in the dataset. Therefore, when the CFG is applied to this particular configuration, none of the embellishment rules apply. In fact, even if the noteset consisting of notes `P1-2-1`, `P1-2-2`, and `P1-2-3` were used, it would also violate the hierarchy because `P1-2-3` would be considered the parent note of `P1-2-2` in CFG terms. In this case, the "New" rule is the only rule that can add an arbitrary interval to another, so that rule is applied and the parse tree for these three notes is passed up the hierarchy. The resulting parse tree for the time-span tree of the second four notes in "Grande Valse Brillante" is shown in Figure 4.4b. It is evident that none of the embellishment rules that were selected could be applied to this phrase.

Another problem arises in creating a full parse tree from each set of three notes as described in the pseudocode of the conversion algorithm. The problem is specifically due to the "New" rule. Once a non-integer string (like the left-hand side of the "New" rule, `N`) replaces an interval symbol in the parse tree, no embellishment rule can be applied above that particular node, because the embellishment rules all have integer interval values on the left-hand side of the rule. An example of this case occurs when the tree-conversion algorithm is applied to Chopin's "Grande Valse Brillante". Each

grammar rule applied at a particular branch is combined with its parent's grammar tree; The tree produced is an invalid parse tree. The tree is shown in Figure 4.5 with the offending branches highlighted. The only solution to this would be to create a rule for every combination of two intervals, as opposed to using the `N` $\rightarrow$ `N N` rule. If the same range of intervals is used—from negative twenty-four to twenty-four—then the ruleset would be expanded by $49^2 = 2401$ rules. Such a high number of production rules would make parsing input sequences extremely slow, and would also create a large amount of rules with zero probability. With a lot of zero-probability rules, there would be many situations where the entire input sequence has a zero probability.

The particular reason that the parse tree in Figure 4.5 is invalid is in the `m3` $\rightarrow$ `m1 N` rule found on the right side of the tree. A minor third interval requires integer intervals as children and cannot incorporate any intervals of arbitrary size, which is what the `N` non-terminal represents. The reason this occurs in the converted tree is precisely because the algorithm looks at the notes that are attached at each branch, so it can compute the intervals exactly, as opposed to incorporating the intervallic information of the child non-terminals as a CFG would. Then the root node from the trees of the lower branches are inserted into the child nodes of the grammar parse from the current branch configuration. Unfortunately, once a `N` non-terminal is used in a tree, the actual interval that created that non-terminal is lost. Since the algorithm computes the intervals from the notes in the current branch configuration, it does not incorporate the intervals of the child branches for each note. Therefore the intervals are not cumulative as they would be in the CFG. This approach was chosen because the rules that are applied by GTTM are applied based on the notes, not the intervals, so it is more true to the original theory of GTTM if the intervals are computed directly from the notes, and not from the cumulative intervals from the child branches for each note.

4.3.2. Supervised Learning with the NLTK Toolkit

Once the format of the GTTM data is compatible for a CFG, the probabilities for each right-hand side of each rule must be found through supervised learning. The Natural Language Toolkit (NLTK) provides the tools necessary for learning probabilities from a solution set of trees. As was described in Section 2.1.6, the induction of a PCFG is only a matter of counting up all of the occurrences of each possible right-hand side for each rule, and then dividing by the total number of times that rule is seen. This process creates a probability distribution for each rule. Once the tree format is converted, the training set will consist of a list of rules that were found in each of the solution files. The format makes it simple to induce a PCFG, since each applied rule is already isolated, regardless of its location in a particular tree. The `induce_pcfg` function provided in NLTK's algorithm will directly induce a PCFG from a list of rules. It will aggregate each rule instance in order to create the corresponding PCFG. It does this by traversing a tree and finding every production rule that was applied to create the tree. Errors in conversion will result in a PCFG that has a different ruleset from the originally specified CFG.

Once each rule's probabilities have been calculated, it is necessary to use the PCFG

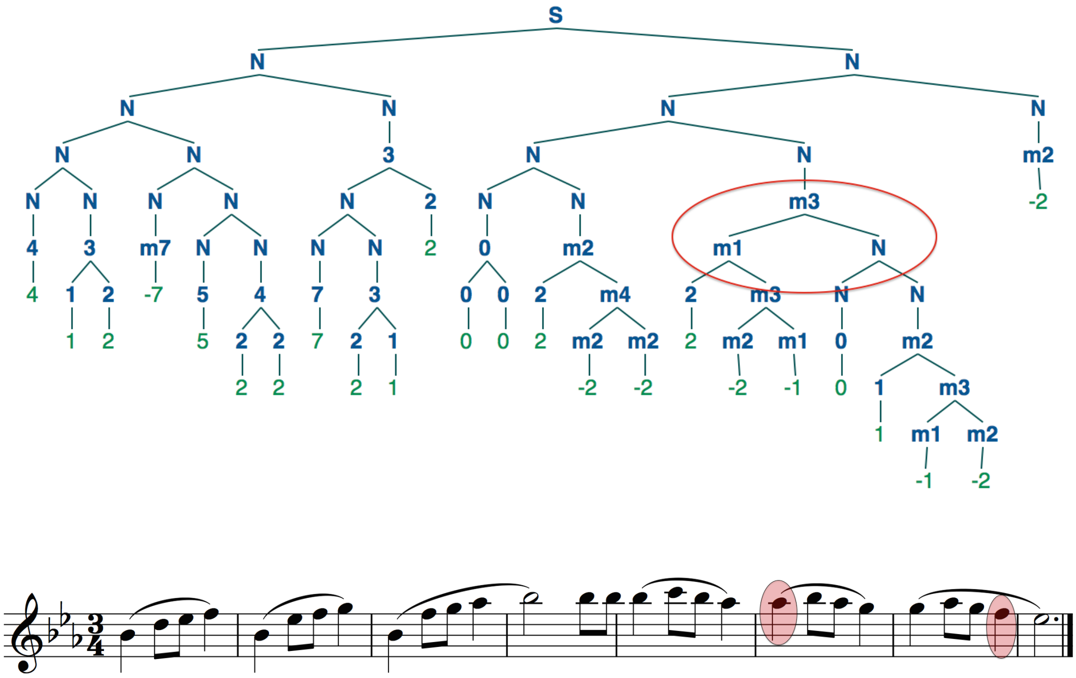


Figure 4.5.: The circled rule does not exist in the grammar that we are using for conversion. The false rule results from applying the grammar at each branch intersection of the solution tree, and aggregating the results from the bottom up. The corresponding score is shown, with the two notes that make up the interval of 'm3' from the circled rule also highlighted.

to parse test cases. NLTK will also be used for parsing a new datapoint with a PCFG. For example, the `ViterbiParser` will return the most probable tree given a PCFG and an input sequence. This work will utilize the `ViterbiParser` in order to find the most probable parse tree for each input melody from the test set.

5. Evaluation Methodology

The approach taken for the evaluation of the Probabilistic Context-Free Grammar (PCFG) for melodic reduction was to compare the entire melodic reduction hierarchy that is output by the PCFG with the full reduction trees given by the analyses created with The Generative Theory of Tonal Music (GTTM). The motivation for comparing trees was that it measures the performance of the model at all levels of melodic reduction. An ideal method should be able to reproduce the reductive decisions both on the musical surface—in the passing of microphrases and melodic figures—and on the deeper levels, where other factors such as compositional form take a larger role. This approach is a strict evaluation strategy since it requires each level to be accurate.

5.1. Constructing the CFG Using Constraints

In order to evaluate the PCFG, one must first design the exact rules that will be used in the underlying Context-Free Grammar (CFG). In Chapter 4, the different musical rules that were chosen to be encoded were described. For example, as was shown for the Escape Tone Rule, the number of CFG rules that are required to expand a single embellishment rule can be large. In order to avoid the human error involved with constructing each CFG rule by hand, a framework was built in order to generate a CFG using constraints and number ranges. The constraint-based system assumes that all of the rules are dealing with integer values.

The software was built in the Python programming language, and has a particular format for each embellishment rule. On the left-hand side of each rule consists of a name with which to identify it (although it will not show up in the final CFG output) as well as a formula for the resulting non-terminal. On the right-hand side are up to two separate non-terminals. Each of these non-terminals have a range associated with it. After the non-terminals is a bracketed list that contains all of the conditions necessary for the rule to be valid. Take for example the constraint-based rule for the Escape Tone Rule described earlier:

```
Escape [i1 + i2] -> i1 [-2:2] i2 [3:24, -3:-24] (i1 * i2 < 0)
```

The formula for computing the resulting value on the left-hand side is given in brackets after the rule's name. Then each right-hand side non-terminal is named, and given any number of ranges, also contained in brackets. Finally, in braces, the conditions that must be satisfied for the rules to be added are given in braces after the right-hand side non-terminals. The naming convention of the non-terminals must reflect the variables given in both the formula for the left-hand side and the conditions on the right-hand

side. Notice that the Escape Tone Rule only requires one condition, `(i1 * i2 < 0)`. This condition ensures that the sign of `i1` and `i2` must be opposite. If they were the same sign, they would either both be positive, resulting in a positive number, or they would both be negative, also resulting in a positive number. The ranges have already specified that `i1` is in the range of a diatonic step, and `i2` is in the range of a diatonic leap. Peculiarly, `i1` also has the value of zero in its range. One would assume that this would allow values that are not considered a diatonic step into the Escape Tone Rule of the final CFG. It is the condition `(i1 * i2 < 0)`, however, that excludes that possibility. Because the condition requires that the product of the two right-hand side values be *less* than zero, and anything multiplied by a zero value is *equal* to zero, then Escape Tone Rules with a zero value are by definition excluded.

The algorithm for expanding the constraints into a string that can be parsed into a CFG is simple:

```
for each constraint-based rule:
    iterate through the first non-terminal's range:
        iterate through the second non-terminal's range:
            iterate through all the necessary conditions, using both the
            first and second non-terminals' current value
                check all conditions for current rule
                if all conditions are met, create the production rule
                string that represents this rule's current configuration
                    expand the left-hand side formula and assign
                    the result to the left-hand side non-terminal
                append the new \gls{cfg} string to the set of all \gls{cfg} rules
    if neither non-terminal on the right-hand side contains a range,
    add the rule as is
```

The algorithm considers every combination of non-terminals within the given ranges, but will only add the rule with the current configurations if and only if every condition is satisfied for the current values.

The result of running the software with only the Escape Tone Rule specified with constraints is:

1 → m2 3	8 → m2 10	15 → m2 17	22 → m2 24
2 → m2 4	9 → m2 11	16 → m2 18	2 → m1 3
3 → m2 5	10 → m2 12	17 → m2 19	3 → m1 4
4 → m2 6	11 → m2 13	18 → m2 20	4 → m1 5
5 → m2 7	12 → m2 14	19 → m2 21	5 → m1 6
6 → m2 8	13 → m2 15	20 → m2 22	6 → m1 7
7 → m2 9	14 → m2 16	21 → m2 23	7 → m1 8

8 → m1 9	23 → m1 24	m9 → 1 m10	m15 → 2 m17
9 → m1 10	m23 → 1 m24	m8 → 1 m9	m14 → 2 m16
10 → m1 11	m22 → 1 m23	m7 → 1 m8	m13 → 2 m15
11 → m1 12	m21 → 1 m22	m6 → 1 m7	m12 → 2 m14
12 → m1 13	m20 → 1 m21	m5 → 1 m6	m11 → 2 m13
13 → m1 14	m19 → 1 m20	m4 → 1 m5	m10 → 2 m12
14 → m1 15	m18 → 1 m19	m3 → 1 m4	m9 → 2 m11
15 → m1 16	m17 → 1 m18	m2 → 1 m3	m8 → 2 m10
16 → m1 17	m16 → 1 m17	m22 → 2 m24	m7 → 2 m9
17 → m1 18	m15 → 1 m16	m21 → 2 m23	m6 → 2 m8
18 → m1 19	m14 → 1 m15	m20 → 2 m22	m5 → 2 m7
19 → m1 20	m13 → 1 m14	m19 → 2 m21	m4 → 2 m6
20 → m1 21	m12 → 1 m13	m18 → 2 m20	m3 → 2 m5
21 → m1 22	m11 → 1 m12	m17 → 2 m19	m2 → 2 m4
22 → m1 23	m10 → 1 m11	m16 → 2 m18	m1 → 2 m3

The last if statement in the pseudo-code also allows for regular CFG rules to be directly inserted into the final CFG. This allows for the transferral of the "New" rule and the "Start" rule from the constraint-based ruleset into the finalized CFG ruleset. Also, as the software runs, the non-terminals that are created on the right-hand side of each rule are saved. For each one of these non-terminals, there will need to be a grammar rule for resolving the non-terminal with its equivalent string. This process happens at the end of any CFG generation, so that all possible non-terminals are included and resolved.

With this software system for generating a CFG string from a set of constraint-based rules not only eliminates human error from the process and saves time, it also allows for the future use of the system for anyone interested in the process. The full specification for the constraint-based grammar, in the new format defined above is:

```
S [] -> N []
N [] -> N [] N []
N  [] -> i1 [-24:24]
Neighbor [i1 + i2] -> i1 [-2:-1, -1:2] i2 [-2:2] {i1 == (i2 * -1)}
Passing [i1 + i2] -> i1 [-2:2] i2 [-2:2] {abs(i1 + i2) >= 3 }
Escape [i1 + i2] -> i1 [-2:2] i2 [3:24, -3:-24] {(i1 * i2 < 0)}
Cambiata [i1 + i2] -> i1 [3:24, -3:-24] i2 [-2:2] {(i1 * i2 < 0)}
```

The full generated CFG grammar string can be found in Appendix C. Only the Neighbor, Passing, Escape Tone, and Cambiata Rules represent embellishment rules—the "New" rule and the "Start" rule are merely necessary to cover all situations, and will not be considered embellishments.

5.2. Tree Comparison

After the CFG rules are formed, the PCFG can be created with supervised training. In order to properly evaluate the efficacy of the trained PCFG, a comparison algorithm must be created in order to validate the parse trees created by the PCFG against the solutions from the GTTM dataset.

The simplest tree comparison algorithm involves comparing the labels at each node in the tree, from the top-down (Hoffmann and O'Donnell 1982). This assumes that the root of each tree (the top-most node) should be the same starting point. The algorithm iterates down through each child node, recursively comparing each label. When a label does not match, that particular branch will be terminated; the child nodes will not be considered for label comparison. This algorithm will find all of the branches that were parsed correctly, starting from the top. For the application to this experiment, there are two main problems with approach. The first incompatibility is with the approach; this method ignores potentially valid parses on the lower parts of branches, and penalizes an entire branch for one single error towards the top. The second problem is in applicability. The top-down comparison algorithm assumes that the roots of the trees are the same, which will not necessarily be true when comparing our parses with our solution trees. In fact, the trees in the solution set are rooted from the bottom up—that is, the same notes are used as input in the solution trees as they are in the parsed result set. Therefore, there should be at least some similarity in the set of leaf nodes in each tree.

The solution trees will sometimes contain branches between notes that are not directly adjacent, so there could be some mismatch between the leaf node sequences. For example, sometimes there are larger intervals missing in the solution between two larger sub-trees because only the notes that were higher in the GTTM tree were compared, as opposed to the notes that were adjacent in the score. In this case, that solution tree would be misaligned with the parse tree by one interval. In order to avoid getting false negatives throughout the rest of the leaf sequence, an alignment algorithm was used so as to match the correct parse tree leaf with the correct solution tree leaf.

Since some of the leaf nodes could be at different indices between the two trees, the process was more complicated than simply iterating through the leaf node set in both the parse tree and the solution tree. It was necessary to first align both sequences of leaf nodes to find the correct match for each leaf node. The alignment algorithm was also borrowed from field of Natural Language Processing (NLP). There is a string-matching algorithm for comparing and analyzing the edit distance between two strings (Levenshtein 1966). The edit distance is defined as the number of edit operations that are required to transform one string into another, including *insertion, deletion*, and *substitution* of characters. This algorithm, once applied, also provides the path between

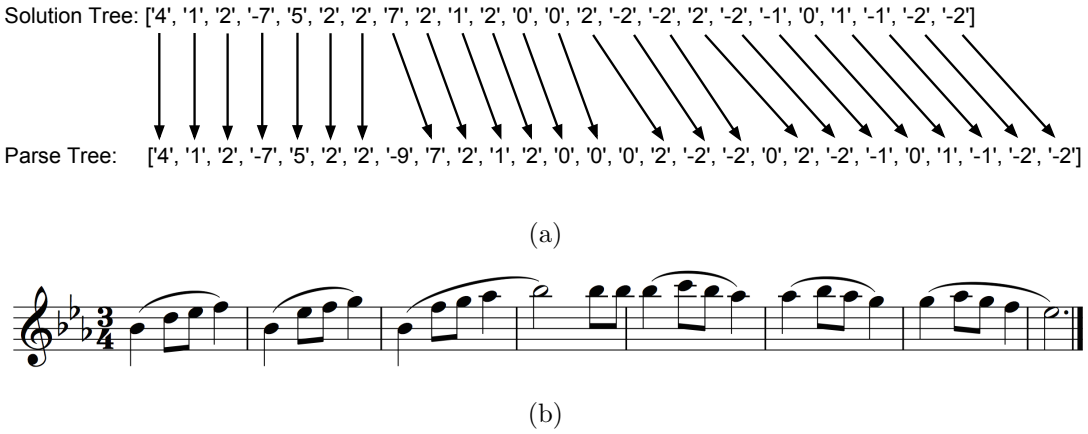
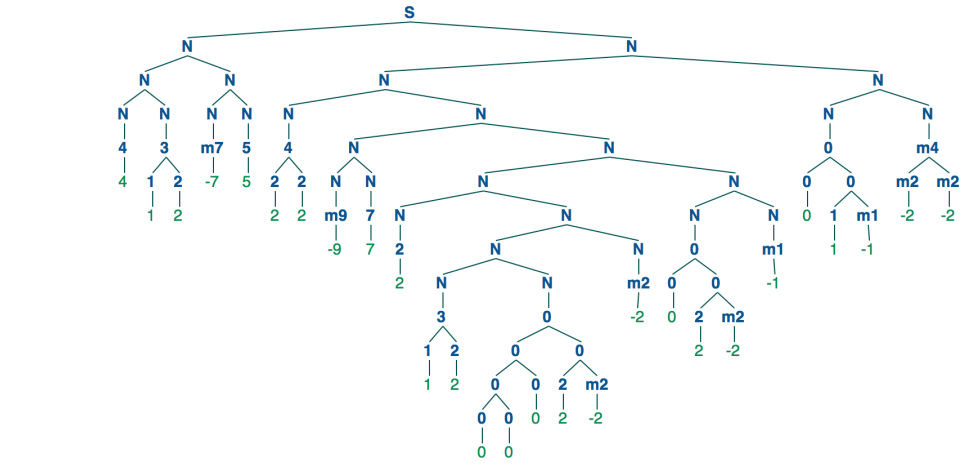


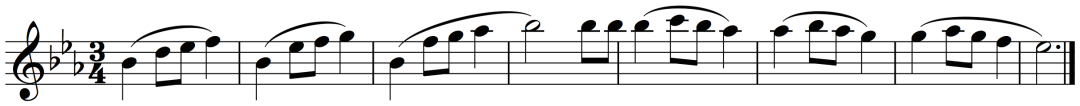
Figure 5.1.: (a) The alignment of the leaf sequence of a prolongational reduction tree with the leaf sequence of the parse tree generated from the trained PCFG. The song used was Chopin's "Grande Valse Brillante", its score shown in (b).

the two strings that align the indices of both character sequences in such a way that maximizes the amount of common elements. This process of aligning leaf nodes with the ground truth solutions was also performed by Granroth-Wilding (2013) in his work using Combinatory Categorial Grammars for harmonic music analysis. An example of the alignment algorithm is given for the first data point in the GTTM dataset. Figure 5.1 displays the alignment between two different sets of leaves—the leaves of the trees of Figure 4.5 and Figure 5.2.

Once the best alignment is found between the solution tree leaves and the parse tree leaves, the branch that contains each leaf is compared from the bottom, up. The algorithm iterates upwards on each tree branch, comparing the solution tree branch with the parse tree branch. All of the non-terminals parent nodes that are the same between the two trees are counted and then marked, so that the nodes cannot be counted twice. This could happen in the case where two consecutive leaves are children of the same parent. That parent could be considered twice. In order to avoid this, the software changes the label of the node to an `'X'` in the parse tree, which will not match any other non-terminal in the solution tree. Take for example, the parse tree from the PCFG tested on the first fold of prolongational reductions, shown in Figure 5.2. Its corresponding solution tree, converted from the GTTM dataset, is seen in Figure 4.5. After the comparison algorithm, the nodes that exist in both the solution tree and the parse tree are marked with an `'X'` as in Figure 5.3. The alignment algorithm allows the comparison of leaves with different indices. In this case, index seven, fourteen, and eighteen in the parse tree are skipped, so that the maximum amount of indices can match. These indices represent intervals of notes that were not directly compared in the solution set, because the hierarchy of the solution had those notes at a different level.



(a)



(b)

Figure 5.2.: A parse tree (a) from the PCFG created from supervised training of the first fold configuration of the prolongational reduction trees found in the GTTM data. The input score was Chopin's "Grande Valse Brillante", its score shown in (b).

There is one additional condition for the equivalence of a node across the two trees being compared. Since the desired outcome is to identify those parts of trees that are structurally identical, the branching must also be considered. Each node has a maximum of one parent, and a maximum of one sibling. In the comparison algorithm, equivalence is not assigned to two nodes that simply have the same label, but also have the identical branching in their parent group. That is, if two nodes have the same label, but one node branches to the right from its parent and the other to the left from its parent, those nodes are not equivalent.

5.3. Cross-fold Validation

The dataset was partitioned into five separate folds. A random number generator was used in order to select random data points for each of the partitions. The PCFG was then trained on a combined set of four out of five partitions, and tested on the remaining set. This process iterated over the five possible folds, each time using a new test set. To generate the solutions, the tree conversion algorithm was used in order to convert

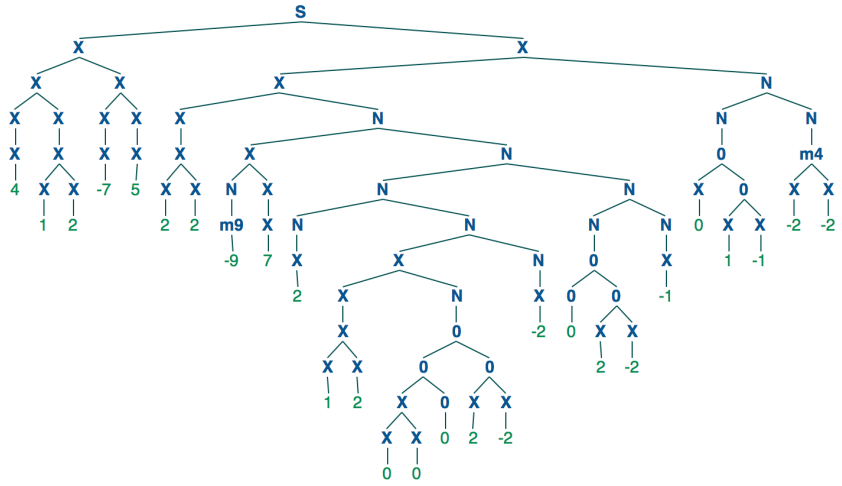


Figure 5.3.: The parse tree from Figure 5.2 after marking the nodes that are equivalent to the similarly-located nodes in Figure 4.5.

the GTTM trees into the proper format for use in the tree comparison algorithm. The tree comparison algorithm was used to compare the output of the trained PCFG with the dataset solutions. The tree comparison resulted in a percentage of nodes that were correctly labelled, as compared with the total number of productions that existed in each tree solution.

There were two sets of possible solutions that the GTTM dataset provided—the prolongational reduction trees, and the time-span trees. As discussed previously, the prolongational reduction trees are created with a process that was informed of harmonic context, while the time-span trees are directly affected by the rhythmic and metric grouping. Another difference was the volume of data. The prolongational reduction trees consisted of one hundred expert solutions while the time-span trees amounted to three hundred. The CFG presented in this thesis does not rely on harmonic information in order to apply the embellishment rules, so theoretically the time-span trees would be more appropriate as a source of input. On the other hand, harmonic data may organize the branches of a melody in such a way that lends itself more to the embellishment rules that were defined in the CFG (this was the case for the small example provided in Figure 4.4 from Section 4.3.1). In order to discover the most appropriate data source, both types of input trees were tested using cross-fold validation.

During the testing process, certain of the test sets had terminals that were not covered by the given training set. For this purpose, "coverage" of a test melody means that the non-terminals encountered within each branching configuration had a non-zero probability. In order for an interval non-terminal to have a non-zero probability, it must be encountered at least once in the training set. The workaround that was created for this situation was to modify the seed value for the random number generator, so that each fold had a new combination of random data points. This process continued until a

configuration was found wherein none of the folds contained a test set for which any of the terminals had a zero probability.

6. Experiment

This chapter details the experiment to evaluate the performance of the Probabilistic Context-Free Grammar (PCFG) for melodic reduction. First, the experiment is described in detail, including the implementation for each step. It is important to consider the details of the implementation, because each decision made in the process of implementation will have an effect on the outcome of the experiment. Once all of the implementation details are uncovered, the tabulated results are shown. Then a baseline for comparing the results will be proposed. Using the parse trees that result from the application of the PCFG, melodic reductions will be presented. With these, the theoretical validity of reduction decisions that the model makes can be evaluated on a case-by-case level.

6.1. Implementation

The software for this thesis was implemented entirely in the Python programming language, using the PyCharm Integrated Development Environment. The code is publicly available[1].

The software is structured into multiple runtime environments, for the tasks of Pre-processing, Training, Cross-fold Validation, Tree Comparison, Creating and Displaying Melodic Reductions From Trees, and Melodic Generation. Each have their own modules for different facets of the task, separated into different source files. This section will describe the implementation in detail for each of the tasks listed. It is designed to walk through the codebase in detail, and the structure of the section will follow this hierarchy:

CFG Construction

constraint_grammar.py builds the rules of the initial CFG in string format, using the defined constraints.

Pre-processing

music_grammar.py converts the input tree data from the format given by The Generative Theory of Tonal Music (GTTM) dataset to the format needed to train a PCFG.

Training, Cross-fold Validation, and Tree Comparison

music_rule_test.py controls the flow of the evaluation process and contains the functions for training the PCFG and performing the cross-fold validation.

[1] https://github.com/bigpianist/PCFGMelodicReduction

music_grammar.py (seen also in the Pre-processing task) collects all the production rules from the input GTTM data, so that the probabilities can be modelled.

validate_tree.py performs the comparison of the parse trees created by the trained PCFG with the converted GTTM solution trees.

Creating and Displaying Melodic Reductions From Trees

score_to_tree.py provides the algorithms necessary to truncate trees by depth, and also to display melodies using either the GTTM solution tree or PCFG tree formats.

Melodic Generation

pcfg_generate.py will generate a randomly sampled tree from a PCFG. It can also selectively sample embellishment rules in order to randomly embellish a melody.

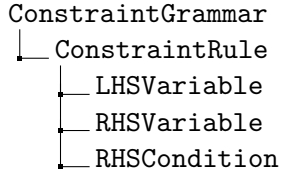
generate_arpeggio.py will generate random output sentences from a given Context-Free Grammar (CFG), and contains the string-based version of the harmony grammar.

6.1.1. CFG Construction

To create a trained PCFG, the process involves the design and implementation first of a CFG, and then the application of a training method in order to model the probabilities of each rule. The methodology for the design of the CFG was presented in Section 4.2. This section details the implementation.

The Natural Language Toolkit (NLTK) was used for the training and parsing of input melodies, as well as the display of resulting parse trees (Loper and Bird 2002). NLTK provides an object-oriented structure for the creation and manipulation of CFGs and PCFGs. The objects are named "CFG" and "PCFG", respectively. Therefore, when designing a CFG, the resulting output must be compatible with the necessary input for forming a CFG object with NLTK. NLTK provides a method for constructing CFGs from strings, which is the method used in this software.

An object-oriented hierarchy was created in order to process a constraint-based grammar and to generate the corresponding CFG in string format. The goal of this grammar creation code is to create a set of integer-based rules automatically from a set of integer ranges and mathematical conditions. In order to do so, the following objects were defined in the *constraint_grammar.py* file of the PCFGMelodicReduction repository:



It is best to show the method of implementing rules with an example. The following is the code for the creation of the Escape Tone Rule:

```
# Escape-tone Rule
left_side = LHSVariable(lambda i1, i2: i1 + i2 )
var1=RHSVariable('i1', [list(range(-2,3))])
var2=RHSVariable('i2', [list(range(-24,-2)), list(range(3,25))])
rs = []
rs.append(var1)
rs.append(var2)
conditions = [ lambda i1, i2: (i1 * i2) < 0 ]
escape_rule = ConstraintRule(left_side, right_side, conditions)
rules.append(escape_rule)
```

It is logical to describe the object-oriented system from the bottom up. A RHSCondition specifies the conditions required for the right-hand side variable to constitute a valid CFG production rule. In the above code, for example, the condition is `(i1 * i2) < 0`. This means that the product of the integer values of the 'i1' variable and the 'i2' variable must be less than zero. The two right-hand side variables have a set of possible integer values, defined by their ranges. When creating all the possible production rules from this ConstraintRule object, the algorithm will iterate through all of the RHSVariables' integer sets, creating every possible combination of integers. Each combination will be tested with the conditions specified in the RHSCondition object. If the current combination of integers satisfies the RHSCondition object's conditions, then a string will be generated that represents the CFG production rule with the current configuration of right-hand side integer values.

For the Escape Tone Rule, specified above, there are two RHSVariables. The first RHSVariable specifies the set of integer intervals, in semitones, that can be considered a diatonic step. The second RHSVariable specifies a diatonic leap. Note that the range values specified are exclusive, not inclusive, so, for example, `range(-24,-2)` is actually the set of all integers from -24 up to and including -3.

The LHSVariable merely specifies the mathematical formula for generating the left-hand side non-terminal of the corresponding CFG rule. In the code example above, the formula is simply to add the two right-hand side variable. An example of a valid CFG rule, specified by the code above is the right-hand side values of -1 and 22. The corresponding CFG string would be `'21 → -1 22'`. This represents one possible manifestation of the Escape Tone Rule.

The ConstraintGrammar specified in the *constraint_grammar.py* file contains the corresponding objects for the Neighbor Tone Rule, the Escape Tone Rule, the Cambiata Rule, the Passing Tone Rule, and the Repeat Rule. For each rule, the same process is applied: iterate through all RHSVariable integer combinations, test the RHSCondition object's conditions (there can be multiple), and generate the CFG string if the conditions are satisfied. The resulting set of CFG production rules is shown in Appendix C.

6.1.2. Pre-processing

Before delving into the exact implementation of the evaluation, one must understand the pre-processing that was required before the CFG could be trained and converted into a PCFG. As described in Section 4.3, the data that was used for the training of the PCFG required a format change before the cross-fold validation could be performed. This section will describe both the dataset used, as well as the pre-processing that was required before the training and evaluation of the PCFG could be performed.

The GTTM dataset

The GTTM dataset (Hamanaka, Hirata, and Tojo 2007b) provides expert annotations of melodic reductions. The data is partitioned into multiple separate files, representing the different theories of GTTM: metrical structure analysis, grouping structure analysis, time-span reduction, and prolongational reduction. Each analysis is contained in its own Extensible Markup Language (XML) file representing the hierarchical data of that particular GTTM analysis. Recall, from Section 2.2.1 that each of the four GTTM components is hierarchical. The original melodies are also provided in MusicXML format (Good 2001). The melodies were chosen from the Western Classical music repertoire of the 18th and 19th century, and each consist of an 8-bar monophonic excerpt from a particular piece. Not every type of analysis was provided for each melody in the dataset. The dataset consist of 300 files in total. For the 300 files, the analyses for the processes of metrical structure analysis, grouping structure analysis, and time-span reduction are all available. For a smaller set of 100 files, every GTTM process has a corresponding analysis. The difference is that the prolongational reduction analyses are only available for the first 100 melodies of the dataset, and not for the remaining 200. The data was obtained through Masatoshi Hamanaka's website[2].

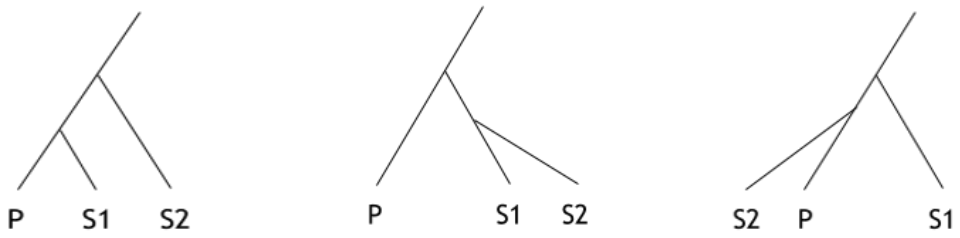
For this thesis, the only GTTM analyses utilized were the time-span reduction and the prolongational reduction. For the time-span reduction data, there were 300 analyses, and for the prolongational reduction analyses there were 100. The format of each of these analysis files is discussed in Section 4.3.1, and an example of a snippet from a time-span reduction analysis file is given in Appendix B.

Converting the Tree Formats

Also described in Section 4.3.1, the key difference in the format of the CFG trees and the format of the GTTM trees is that the CFG rules require a set of three notes in order to utilize a melodic embellishment rule. In order for a note to be *subsumed* in the GTTM trees, only one other note is required. Because of this, it was necessary to convert between the two formats.

An algorithm for traversing a GTTM tree and creating the corresponding CFG tree was created in the *music_grammer.py* file. Within that file, a function was designed for recursively iterating through the XML file for the given GTTM tree, and applying

[2] http://www.gttm.jp/ accessed April 22, 2014.



(a) (b) (c)

Figure 6.1.: Example of the three possibilities for the temporal sequences of the different GTTM branching. In these examples, 'P' stands for the `primary` tag, 'S1' stands for the first `secondary` tag, and 'S2' stands for the second `secondary` tag.

the CFG at each branching location. The resulting parse tree that was created at each branching location by the CFG was then accumulated and appended recursively, so that a full parse tree was created that represented the melodic reduction CFG applied at each branch.

As presented in Section 4.3.1, each branching location of the GTTM tree data is encoded with a `primary` tag and a `secondary` tag. Each of these tags refer to a particular note in the melody. The note from the `secondary` tag is *subsumed* by the note in the `primary` tag. In order to apply the CFG at a given branch, however, three notes are required. Therefore, one must not only have the current `primary` and `secondary` tags—rather, one must find the closest *sibling* of the `secondary` tag by iterating up the tree, and finding the next-highest `secondary` tag.

Once a set of the three closest notes in the current tree is found, the order of the notes must be validated. Figure 6.1 shows the different possibilities for a left-oriented primary branch. In these examples, 'P' stands for the `primary` tag, 'S1' stands for the first `secondary` tag, and 'S2' stands for the second `secondary` tag. The CFG requires a sequence of three notes in order for the application of embellishment rules. However, if the note from the `primary` branch is in the center of the set of three notes, the embellishment rule must not be applied, since the `primary` branch *subsumes* both of the other branches. In that case, Figure 6.1a is the only valid branching configuration for applying the CFG rules. In the case of the branching configurations shown in Figure 6.1b and Figure 6.1c, the only CFG rule that can apply is the "New" interval rule, since that allows for any interval.

The algorithm developed for converting the GTTM tree into the corresponding CFG tree was required to first check the temporal order of the notes before applying the grammar. In the corresponding pseudocode below, the *isOrderValid* function defines whether or not the embellishment rules can be applied. The algorithm performs a depth-first search, gathering sets of the nearest three notes and applying the CFG as

appropriate. At each level, both the note and the resulting parse tree for both the `primary` and the `secondary` branches are passed up to the next-highest branch location. In the function, this is represented by the return value—it is a tuple containing those values. One other thing to note is that, if there is a child `primary` tag for a given node, there must also be a `secondary` tag. The pseudocode for the algorithm is displayed in Algorithm 1.

Algorithm 1 Get CFG Tree From GTTM Tree Recursively

```
1: procedure applyCFG(note1, note2, note3)
2:     if isOrderValid(note1, note2, note3) then
3:         fullTree ← getCFGTree(note1, note2, note3),
4:     else
5:         tree1 ← getCFGTree(note1, note2),
6:         tree2 ← getCFGTree(note2, note3),
7:         fullTree ← getCFGTree(tree1, tree2),
       return fullTree
8: procedure parseBranch(head, musicXml, CFG)
9:     if primary in head then
10:        primaryTuple ← ParseBranch(head.primary, musicXml, CFG)
11:        secondaryTuple ← ParseBranch(head.secondary, musicXml, CFG)
12:    if primaryTuple then
13:        note1 ← primaryTuple.primaryNote
14:        note2 ← primaryTuple.secondaryNote
15:        note3 ← secondaryTuple.primaryNote
16:        primaryTree ← applyCFG(note1, note2, note3)
17:
18:        primaryTree ← mergeTree(primaryTree, primaryTuple.primaryTree)
19:        primaryTree ← mergeTree(primaryTree, primaryTuple.secondaryTree)
20:    if secondaryTuple then
21:        note1 ← primaryTuple.primaryNote
22:        note2 ← secondaryTuple.primaryNote
23:        note3 ← secondaryTuple.secondaryNote
24:        secondaryTree ← applyCFG(note1, note2, note3)
25:
26:        secondaryTree ← mergeTree(secondaryTree, secondaryTuple.primaryTree)
27:        secondaryTree ← mergeTree(primaryTree, secondaryTuple.secondaryTree)
28:    newPrimNote ← primaryTuple.primaryNote
29:    newSecNote ← secondaryTuple.primaryNote
30:    newTuple ← [newPrimNote, primaryTree, newSecNote, secondaryTree]
31:    return newTuple
```

6.1.3. Training

NLTK provides the necessary functionality to train a PCFG. The `induce_pcfg` function will automatically build the corresponding PCFG object by simply giving a list of all the production rules that were seen in the set of solution parse trees used for training. The function requires that the production rules be isolated so that the individual parse tree created at each branch is listed separately. This is the perfect format for the the converted GTTM trees, because the trees were converted by applying the grammar at each branching position! Therefore, the previous algorithm for converting the GTTM trees into CFG trees (seen in Algorithm 1) was re-used, with the simple addition of collecting a list of individual parse trees that were created at each branch. The pseudocode for the resulting algorithm is shown in Algorithm 2. Note that the `productionList` parameter to the `parseBranch` function is passed by reference, so the calling function would allocate that variable and it would be automatically modified and returned. The production list is populated before the trees are merged with the trees from the lower branches, on lines 17 and 26 of Algorithm 2, which ensures that the production rules added represent only the current branching configuration. The updated conversion algorithm, when applied to a particular GTTM solution, would return the following:

1. A single CFG tree representing the combined production rules created by applying the CFG at every branch of the GTTM tree, to be used for evaluation.

2. A list of all the individual production rules that were applied at each branch, to be used for training.

This process could be applied to both the time-span trees and the prolongation trees.

Once the production list is created that aggregates all of the individual CFG trees from every branch of every input GTTM tree, the creation of the trained PCFG was as simple as calling the `NLTK.induce_pcfg` function with only the starting production rule, and the list of productions as parameters.

6.1.4. Tree Comparison

The tree comparison method is what defines the evaluation for a particular parsed melody. The CFG trees resulting from the conversion of the GTTM reductions are the ground truth, so the task is then to compare the parse trees created with the trained PCFG with the converted GTTM trees. The methodology for this comparison was described in detail in Section 5.2, and can be summarized thusly:

- Select the sequence of leaf elements from both trees

- Align the leaf sequences to maximize identical leaves between the two sets

- For each aligned pair of leaves, iterate up the tree, comparing production rule nodes

Algorithm 2 Collect All Productions From GTTM Trees

```
1: procedure applyCFG(note1, note2, note3)
2:     if isOrderValid(note1, note2, note3) then
3:         fullTree ← getCFGTree(note1, note2, note3),
4:     else
5:         tree1 ← getCFGTree(note1, note2),
6:         tree2 ← getCFGTree(note2, note3),
7:         fullTree ← getCFGTree(tree1, tree2),
       return fullTree
8: procedure parseBranch(head, musicXml, CFG)
9:     if primary in head then
10:        primaryTuple ← ParseBranch(head.primary, musicXml, CFG)
11:        secondaryTuple ← ParseBranch(head.secondary, musicXml, CFG)
12:    if primaryTuple then
13:        note1 ← primaryTuple.primaryNote
14:        note2 ← primaryTuple.secondaryNote
15:        note3 ← secondaryTuple.primaryNote
16:        primaryTree ← applyCFG(note1, note2, note3)
17:        productionList.append(primaryTree)
18:
19:        primaryTree ← mergeTree(primaryTree, primaryTuple.primaryTree)
20:        primaryTree ← mergeTree(primaryTree, primaryTuple.secondaryTree)
21:    if secondaryTuple then
22:        note1 ← primaryTuple.primaryNote
23:        note2 ← secondaryTuple.primaryNote
24:        note3 ← secondaryTuple.secondaryNote
25:        secondaryTree ← applyCFG(note1, note2, note3)
26:        productionList.append(secondaryTree)
27:
28:        secondaryTree ← mergeTree(secondaryTree, secondaryTuple.primaryTree)
29:        secondaryTree ← mergeTree(primaryTree, secondaryTuple.secondaryTree)
30:    newTuple ← [primaryTuple.primaryNote, primaryTree, secondary.primaryNote,
   secondaryTree]
31:    return newTuple
```

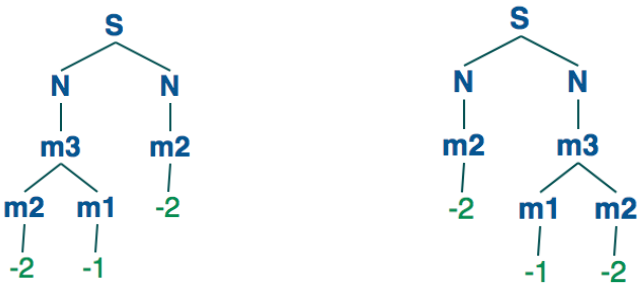


Figure 6.2.: The trees shown in (a) and (b) have no nodes in common. A node is considered to be only a production rule, not a leaf (e.g., `m2` but not `'-2'`). Each node must also branch from its parent in the same way as the corresponding node in the compared tree.

- For each production rule node, the label as well as the branching structure must be identical

A production rule node is a node that represents the left-hand side non-terminal for the rule that was applied at the current branch. The leaves themselves are not considered part of the comparison, since any leaf that is correctly identified will have at least one corresponding production rule that represents the same information. Therefore, having leaf node that is correctly aligned between two trees (say `'-2'`), is not the same as having an equivalent tree node (an `m2` node with the correct branching). More simply put, the leaves are not considered part of the parse tree for comparison purposes.

The last step in the comparison method is critical. There can be nodes that are identically labelled, but if they do not constitute the same tree formation, then the trees are not identical at that node. For example, consider the leaf sequence `[-2, -1, -2]`. Two trees that non-exhaustively represent the different parsing possibilities are shown in Figure 6.2. With the PCFG that was created for this research, there are only three possible branching configurations for a given node: left child, right child, and single child. Using the defined method of comparison, these two trees have no nodes in common. The first `m2` node in Figure 6.2a, for example, is a left-child node. Comparing that node with the first `m2` node in Figure 6.2b shows that the node labels are identical (`'m2'`), but the branching is distinct; the `m2` node in Figure 6.2a is a left-child node, while the the `m2` node in Figure 6.2b is a single-child node. Therefore these two trees cannot be considered equivalent at that particular node in the tree. The same branching issues arise with the other leaves' parent nodes.

Comparison Implementation

The leaves of the two trees to be compared are first aligned. The algorithm that performs this alignment is based on dynamic programming, and was originally developed as a

method to compute the edit distance between two strings (Levenshtein 1966). The Levenshtein distance computes the minimum number of edit operations that are required to convert one string into another. It allows the edit operations of insertion, deletion, and substitution. Each action has a particular cost associated with it. In order to adapt this to two sets of leaf sequences, the substitution operation had to be removed. The leaf sequences cannot be changed, therefore substitution was not an option. In order to implement the Levenshtein algorithm, a matrix is created for which one dimension is the length of one string plus one, and the other dimension is the length of the other string plus one. Each operation has a direction associated with it. In each cell, the algorithm is comparing how to change one string into the other, using the operations. The modified pseudocode is shown in Algorithm 3. Given the matrix returned from this function, it is possible to know which indices from one list should be removed in order to create the sequence from another list. One can then trace the minimum value path back from the bottom-right cell, getting a path that determines the best alignment. The best alignment guarantees the the maximum number of elements between the two sets of leaf nodes will be identical.

Once the best alignment is discovered among two sets of leaf nodes between two trees, the different branches that contain those leaves can be compared between the trees. The software iterated through all paired sets of leaf nodes, and then applied a bottom-up tree comparison for every pair of branches. The pseudocode for the bottom-up tree comparison is given in Algorithm 4. The comparison computes the number of identically-formed nodes in the branches from two different trees. As was mentioned in Section 5.2, a node is identically formed when both its label and its placement as compared with its sibling node are equivalent. With the combinations of these algorithms, the total number of identically-formed nodes for a pair of trees can be discovered.

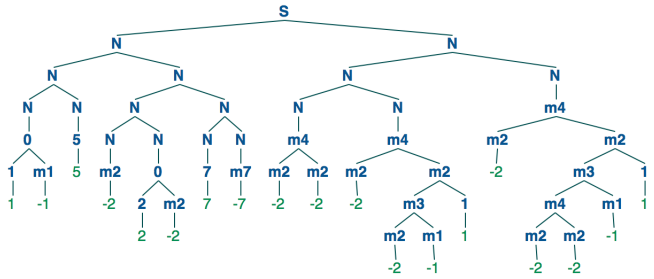
Figure 6.3 shows the comparison algorithm in action. Figure 6.3a shows the solution tree converted from the GTTM time-span tree data. Figure 6.3b is the most probable parse tree, after training the PCFG, and testing it on the fourth fold of the time-span tree data. The alignment algorithm is then applied to the leaves of both trees, giving the alignment that represents the maximum number of similar leaf elements between the two trees. This alignment is necessary because of the intervals that were omitted because of the conversion algorithm, as described previously. In Figure 6.3c, the result of the comparison algorithm is shown. Each node that both has the same label, and is in the same branching configuration between the parse tree and the solution tree is marked with an `'X'`. One can see, for example, the Neighbor Tone Rule is discovered in both trees: first between the first two intervals (`'-1'` and `'1'`) of the song, and then later between the sixth and seventh intervals (`'-2'` and `'2'`). However, above both of these embellishment rules in Figure 6.3c, either the node label changes (as with the parent node `'S'` of the first Neighbor Tone Rule) as compared with the solution tree, or the branching configuration changes (as with the parent node `'N'` of the second Neighbor Tone Rule, which is a right-branching child in the solution tree and a left-branching child in the parse tree).

Algorithm 3 Leaf Alignment Algorithm

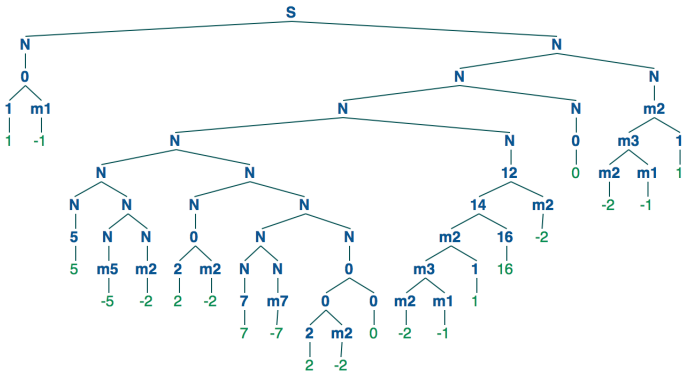
```
1: procedure editDistance(list1, list2, numCorrect)
2:     d ← matrix(len(list1) + 1, len(list2) + 1)
3:     //moving across the matrix represents deletion
4:     //The source list can be transformed into the empty
5:     //list by deleting all indices
6:     for i from 1 to len(list1) do
7:         d[i, 0] ← 0
8:     //moving down the matrix represents addition
9:     //The target list can be reached from the empty
10:    //list by adding all indices
11:    for j from 1 to len(list2) do
12:        d[0, j] ← 0
13:    for j from 1 to len(list2) do
14:        for i from 1 to len(list1) do
15:            if list1[i] == list2[j] then
16:                d[i, j] ← d[i − 1, j − 1] //no operation
17:            else
18:                deletion = d[i − 1, j]
19:                addition = d[i, j − 1]
20:                d[i, j] ← min(deletion + 1, addition + 1)
21:    return d//minimum distance is in index [m, n]
```

Algorithm 4 Compare Trees Bottom Up

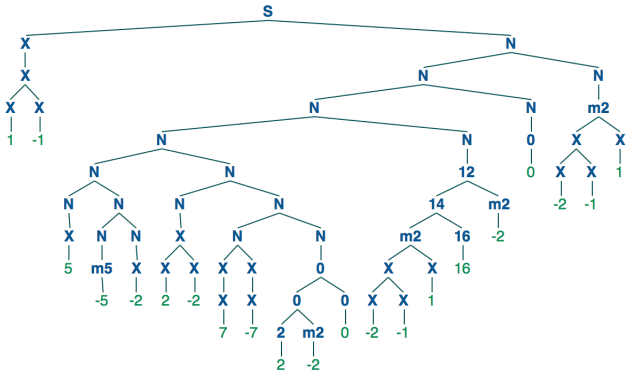
```
1: procedure compareBottomUp(node1, node2, numCorrect)
2:     label1 ← node1.label
3:     label2 ← node2.label
4:
5:     isLeftChild1 ← isNodeLeftChild(node1.parent, node1)
6:     isLeftChild2 ← isNodeLeftChild(node2.parent, node2)
7:
8:     if label1 == label2 and isLeftChild1 == isLeftChild2 then
9:         node1.label ←'X'
10:        numCorrect ← numCorrect + 1
11:        parentNumCorrect ← compareBottomUp(node1.parent, node2.parent, 0)
12:        numCorrect ← numCorrect + parentNumCorrect
13:    return numCorrect
```



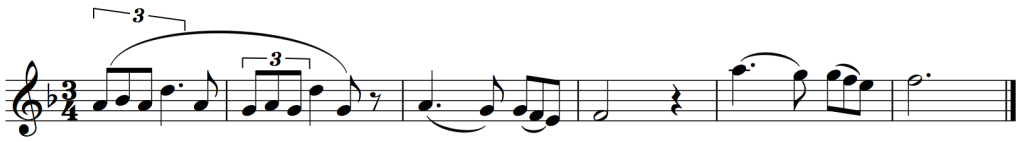
(a) The solution time-span tree gathered from the GTTM dataset, converted into CFG format.



(b) The most probable parse tree using the PCFG trained with the fourth fold configuration of the time-span tree data.



(c) The parse tree from (b), after comparison with the solution time-span tree in (a). The nodes that are equivalent between the two trees are marked with an 'X'.



(d) The original melody for the fourth movement of Schubert's "Schwanengesang".

Figure 6.3.: The comparison process for the melody in Schubert's "Schwanengesang".

6.1.5. Cross-Fold Validation

With the algorithms defined for the conversion of the GTTM trees (Algorithm 1), the training of the PCFG (Algorithm 2 plus NLTK), and the method of comparison (Algorithms 3 and 4), the cross-fold validation is relatively straightforward. The process was as follows for both the time-span tree data and the prolongation tree data:

1. Randomly create 5 equal-sized separate subsets, or folds, of the dataset.
2. Train the PCFG on 4 of the 5 folds.
3. Use the trained PCFG to parse each melody from the remaining test set, then compare each parse tree with the solution tree using the comparison method.
4. Repeat steps 2 and 3 until each fold has been the test set one time.

For step 1, the Python module `random` was used. Five separate subsets were created by randomly selecting melodies from the dataset, in even parts. The corresponding code for the cross-fold validation of the PCFG was placed in the file *music_rule_test.py*. It controls the flow of the process, and also contains the functions required to execute the validation, `generateFoldIndices` and `crossVal`. The implentations for these functions were straightforward so the pseudocode is not supplied.

The percentage of nodes that were identically-formed between the parse trees created with the trained PCFG and their corresponding converted GTTM trees was aggregated of the entire test set, for each fold configuration. Similarly, the sum of all nodes in the converted GTTM trees was also computed, giving a percentage of accuracy for each fold.

6.1.6. Creating and Displaying Melodic Reductions From Trees

It is also important to understand the types of decisions that result from the PCFG, in terms of which notes are reduced. For the trees that result from the PCFG, as well as the original solution trees, it is possible to remove the deepest level of branches in order to subsequently reduce the melody. The GTTM solution trees are generally very evenly distributed. That is, they tend to be not very deep, and to have similar depth on each branch. Both types of the PCFG parse trees, however, can be very unbalanced and also very deep. Therefore, two separate methods were developed to create melodic reduction examples from the GTTM trees and the PCFG trees, respectively.

For the GTTM solution trees, it was as simple as defining a single depth, and traversing every branch to that level, truncating any branches that extended beyond that depth. Because the branches are even, this created the expected result—notes were generally removed from multiple places in the melody in each step. For the parse trees, a different approach was taken. Instead of specifying the depth as a distance from the root, a *negative* depth was assigned. With this method, any node that was within a certain distance of the leaf node *of the current branch* would be truncated. This produced a similar result to the depth-based method with the GTTM trees, even though the forms of the trees tended to be very different. Figure 6.4 shows an example of the

parse tree method. Figure 6.4a is the most probable parse tree for the PCFG trained with the second fold configuration of the prolongational reduction data. Note that the maximum depth is 8, while the branch on the right side has a depth of only 5. Figure 6.4b shows the first step in the negative depth reduction method. After reducing this tree by one level, both the branch of depth 8 and depth 5 are truncated. This process continues one level at a time until Figure 6.4d is produced. The tree in Figure 6.4d can no longer be reduced, because there are no longer any embellishment rules applied in that tree, there is only the "New" rule. As was presented in Section 4.3.1, once the "New" rule is used, the resulting integer interval data is lost. For this reason, the tree can no longer be reduced. The code for the depth-based tree reduction algorithm as well as the negative depth reduction algorithm is provided in the *score_from_tree.py* file. The depth-based tree reduction algorithm is a standard depth-first search with a simple counting mechanism incorporated. Once the appropriate depth is found, the branch is truncated. The negative depth reduction algorithm is more of a custom case. Therefore, the pseudocode for the negative depth tree truncation algorithm is provided in Algorithm 5. The pseudocode provided is slightly simplified from the original version—the original version also includes the tracking of removed note indices so that the notes removed can be identified for the generation of the score at a later time.

Algorithm 5 Negative Depth Tree Reduction Algorithm

```
1:  procedure truncateToNegDepth(node, desiredDepth)
2:      maxChildDepth ← 0
3:      //the depth function finds the depth of the tree from the current node and below
4:      treeDepth ← node.depth()
5:      for child in node.children do
6:          if node.label == 'N' then
7:              child, maxChildDepth ← truncateToNegDepth(child, desiredDepth)
8:          else
9:              doTruncateBranch ← treeDepth <= desiredDepth
10:             branchNotYetTruncated ← (maxChildDepth + 1) <= desiredDepth
11:             if doTruncateBranch and branchNotYetTruncated then
12:                 node.deleteChildren()
13:             else
14:                 child, maxChildDepth ← truncateToNegDepth(child, desiredDepth)
15:     return node, maxChildDepth
```

6.2. Results

There were two different types of reductions that were tested: the time-span trees created from the Time-Span Preference Rules (TSPRs), and the prolongational reduction trees created from the Prolongational Reduction Preference Rules (PRPRs). Cross-fold validation was applied for each, resulting in aggregated statistics on the effectiveness of

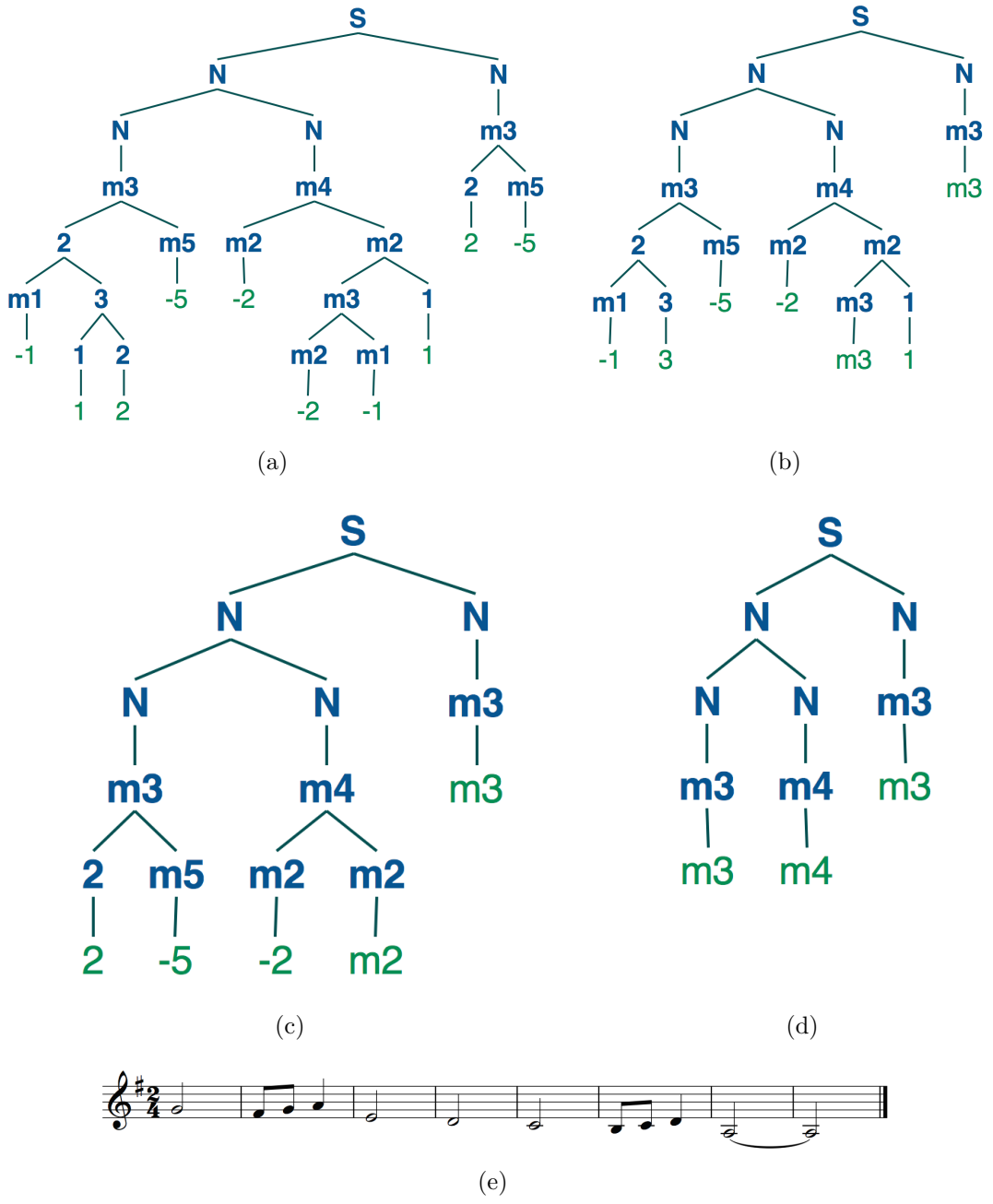


Figure 6.4.: The most probable parse tree, trained with the second fold configuration of the prolongational reduction GTTM data. All of the embellishment rules were removed, at diminishing levels of depth from (a) to (d). The source melody, *Pomp and Circumstance* by Sir Edward Elgar is shown in (e).

each approach for each fold. The results for each fold were then averaged over all of the five different fold configurations.

Type of Tree	Fold Number	Percentage of Matched Nodes
Prolongation Reduction	1	14.16
	2	12.84
	3	14.09
	4	10.72
	5	13.53
Average Percentage	All	13.07
Time-span	1	12.69
	2	15.49
	3	14.76
	4	14.91
	5	17.26
Average Percentage	All	15.02

Table 6.1.: The results from cross-fold validation for each data set.

The statistics resulting from cross-fold validation are shown in Table 6.1. The PCFG matched more closely to GTTM with the time-span tree data. Table 6.1 shows only the total nodes that matched using the bottom-up tree comparison algorithm, after alignment. It is important to consider these percentages as compared with some baseline metric. The leaf sequences are being considered as a starting point for the bottom-up comparison. For these tree datasets, it would make sense to use the comparison of the two leaf sequences as a baseline to judge the success of a model; a model that accurately represents the data set should be able to come close to reproducing the leaf sequence of the solution set. Table 6.2 shows the fold performance over the two types of data as compared to the percentage of leaves that are in the solution set. To compute the percentage of leaves in each solution tree, all of the productions were aggregated over each tree from each fold. Likewise, all the leaves from each tree of each fold were aggregated, and the percentage of leaves per the number of productions was then easily computed. These leaves represent the percentage of nodes in the solution GTTM trees that are leaf nodes of their branch. This baseline was chosen because, ideally, the parse trees should have the same leaf sequence as the solution trees. This is not the case, because an alignment algorithm was required to align the leaf nodes of the PCFG parse trees with the solution GTTM trees. Using this baseline should be the starting point for comparison.

6.2.1. Discussion

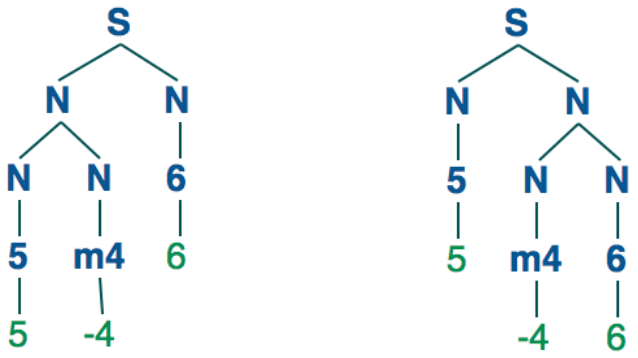
It is clear that the PCFG's performance is worse than the defined baseline metric. For an ideal reduction, the nodes of the leaves between the parse tree and the solution tree would be identical, as well as all of the higher-level nodes in the tree. On average, a given

Type of Tree	Fold	Total Leaves	Total Productions	Leaf Percentage
Prolongation Reduction	1	537	1518	35.38
	2	526	1461	36.00
	3	617	1719	35.89
	4	673	1869	36.00
	5	600	1677	35.78
Average Percentage	All			35.81
Time-span	1	1959	5491	35.68
	2	1894	5365	35.30
	3	1942	5508	35.26
	4	1741	4842	35.96
	5	1768	4976	35.53
Average Percentage	All			35.54

Table 6.2.: The percentage of leaf nodes for each fold, for each data set.

parse tree's leaf nodes that match the corresponding solution tree's leaf nodes, plus their matching reduction nodes at a higher level do not sum to the percentage of leaf nodes in the corresponding solution trees. This means that the reductions are very inconsistent, and the number of situations in which they fail at the surface level, on average, outweigh the number situations in which they succeed. This is likely due to a few different factors. For one, the solution trees do not have the same leaf sequences as the parse trees. A decision was made when converting the solution trees to an input format compatible with a CFG to consider the sets of notes at each branching configuration in the trees created with GTTM, and to compute the intervals from the notes in that set. Accordingly, the intervals are not always computed from the notes that are directly adjacent in the score. This decision was made in order to accurately reflect GTTM, since GTTM based its rules on notes and not intervals. The depiction of this problem can be seen in the comparison of the trees in Figures 4.4a and 4.4b. Figure 4.4a contains the particular configuration that allows for the creation of the leaf sequence from adjacent notes in the input melody, while Figure 4.4b does not, resulting in the duplication of an interval of two semitones. This issue does not, however, affect the correctness or the applicability of the training algorithm, nor does it implicate that the form of the resulting parse trees is incorrect in any way.

Another factor that drops the percentage is simply that the number of possible parse trees with the current CFG ruleset is quite large. For a particular sequence of ten intervals taken from an arbitrary file in the GTTM dataset, for instance, the number of possible parse trees using the original CFG amount to 17,616. Part of the reason why there are so many different tree configurations is because of the rule that allows the repetition of any intervals that fall in the defined range—the `N` $\rightarrow$ `N N` rule. This rule was necessary because not every interval combination is described in the embellishment rules that were chosen. Consider a hypothetical interval sequence of `[5, -4, 6]`, for example. There are no embellishment rules that apply to this particular situation.



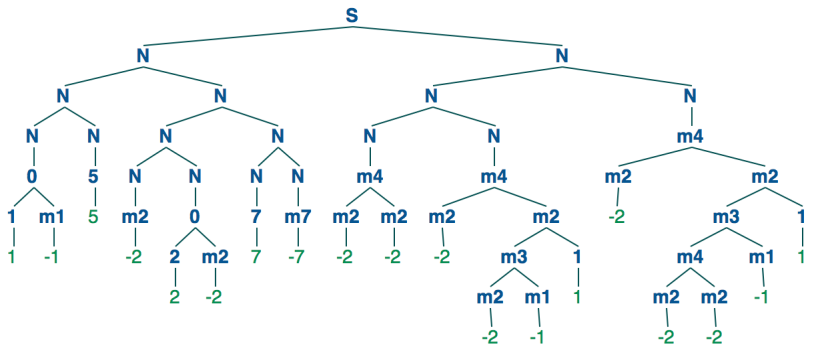
(a) (b)

Figure 6.5.: The trees shown in (a) and (b) are two possible combinations of the same interval sequence, and both provide little information in terms of melodic reduction.

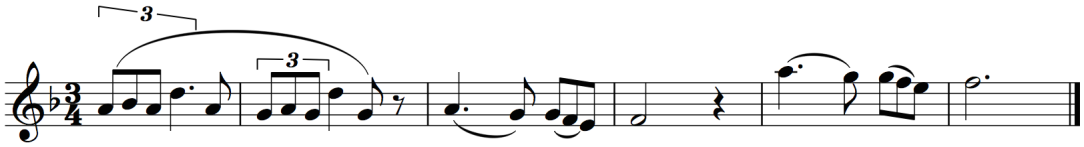
Therefore the "New" rule of `N` → `N N` must be used. With the application of the "New" rule, however, there are still two branching configurations that can be created. The two possibilities are given in Figure 6.5. These two different parse trees do not provide any information as to whether any notes were embellished, or if one note is more important to the melody than another, yet they still represent two different branching configurations.

Lastly, there are no CFG rules that represent the branching that occurs at a higher level in the GTTM trees. The GTTM trees are influenced by musical characteristics such as musical phrasing, song structure, and harmony once the tree reaches its higher levels. The embellishment rules do not take any of these musical features into account, and therefore will fail when those features influence the solution set.

A specific example illuminates the challenges more effectively. Figure 6.6 shows a melody chosen from the GTTM dataset, as well as the corresponding time-span tree converted into the CFG format that is necessary to train the grammar. Immediately, one can see that some of the larger intervals are left out of the solution tree's leaf nodes. Specifically, the fourth interval should really be a jump of `'-5'`. Unfortunately, this particular interval is represented higher in the tree, and there is no embellishment rule that defines it. Therefore, the interval was converted into the `'N'` token and lost from the set of leaves. This happens often for intervals that are between two melodic phrases, which was the case for this interval. Notice, for example, that the `'-7'` interval at the end of the second measure is included in the CFG tree. It is included because it represents an interval that is contained within a single phrase, as opposed to the connection between two phrases. It is evident that for each new subtree, there is a leaf interval that is lost to the `'N'` rule: the `'2'` between notes 10 and 11, the `'0'` between notes 11 and 12, the `'16'` between notes 16 and 17, and the `'0'` between notes 18 and 19. Some of these intervals are candidates for embellishment rules, however the embellishment rules require



(a) The converted time-span tree gathered from the GTTM dataset.



(b) The original melody for the fourth movement of Schubert's "Schwanengesang".

Figure 6.6.: An example time-span tree converted into the CFG format, and the corresponding score.

a particular shape between a sequence of two intervals, and in this case the sequence each interval was a part of did not match any of the embellishment rules. Until every interval movement—including those in the higher-level branches—can be described by embellishment rules that resolve to numeric intervals, the set of leaf intervals will be incomplete.

6.2.2. Analysis of the Reductions

The reductions generated from the trained PCFG are useful to identify the possible reasons that certain reduction decisions made by the algorithm do not match those made by the experts in the solution GTTM trees. Presented in this section are some analyses of the reductions produced by truncating the parse trees and the solution trees at certain depths, as described in Section 6.1.6.

Pomp and Circumstance

Figure 6.7 shows both the PCFG reductions and the GTTM reductions of the melody *Pomp and Circumstance* by Sir Edward Elgar. This figure illuminates a particular weakness in the musicological effectiveness of the given model. In Figure 6.7b, between the original, top melody, and the first level of melodic reduction below it, there is an unlikely rule applied. The last four notes of the original melody are B, C, D, and A, in that order. The algorithm elects to use the `m3` $\rightarrow$ `1 m4` rule on the last three notes. When looking at the score, however, the Passing Tone Rule is much more applicable to

the preceding group of three notes, namely B, C, and D. It is the metrical placement of these notes that determines this preference. The PCFG has chosen to use the note on the second beat of a measure as an anchoring tone. It is much more likely that the first and third beat of the third-to-last measure would be the anchoring tones, because both of those metrical placements are stronger than the second beat. The solution from GTTM in Figure 6.7a is congruent with this synopsis. It seems that having metrical information encoded in the interval representation would definitely help in this situation.

A similar situation occurs between the third and fourth rows of Figure 6.7b, in measures 3–6 with notes E, D and C. It is not likely for a note such as this D, which has an onset on the downbeat of a measure, and a duration of over two measures, to be a passing tone to a note that sounds on the second beat of the measure, and has a much smaller duration. In this case, there are no other embellishment rules that would apply for this group of notes, or their directly adjacent notes. When comparing the melody at levels 3 and 4 in the reduction with the equivalent level of reduction in the solution of Figure 6.7a, it is clear that these "errors" in melodic reduction can also compound as the reduction gets higher in the parse tree; the reductions in Figure 6.7b seem to become more and more disparate from the solution reduction in Figure 6.7a as the reduction depth increases. For the situation of passing tone in measures 3–6, it indeed seems that the mistake is a result of previous reductions, since the passing tone is the only possible rule to apply in that situation.

The parse tree for the melody of *Pomp and Circumstance* created with the PCFG trained on the time-span GTTM tree data is also shown in Figure 6.8. When compared with Figure 6.4a, it is evident that the differences are only superficial. It is only the configuration of the nodes that represent the "New" rule that are different. In fact, it would create the same set of melodic reductions as in 6.7 if the same negative depth pruning method were used.

Für Elise

Figure 6.9 shows an example of another comparison of the the reductions created with a parse tree and the reductions created with a solution tree. In this case, the PCFG was trained on the time-span reduction data. Similarly, the solution file chosen was the time-span reduction tree given by the GTTM dataset (Hamanaka, Hirata, and Tojo 2007b). This is a great example of where the reductive capabilities of the PCFG model fail but also where they succeed. In Figure 6.9a, the second row shows the first phrase being reduced. In the first full measure (as well as the preceding anacrusis), the opening figure of two identical neighbor tones is correctly identified. Following that, an escape tone is found in the D natural note that occurs at the end of the first full measure. Looking at the same staff, in the fourth full measure, when the opening figure repeats, it is evident that the figure is reduced in exactly the same way. One positive thing about the PCFG is that it is likely to reduce an identical passage in the same way, since the probabilities are static after training. In Figure 6.9b, the solution reductions, the two passages are also reduced in almost exactly the same way.

Where the PCFG fails is in the arpeggios in between the iconic opening figure and

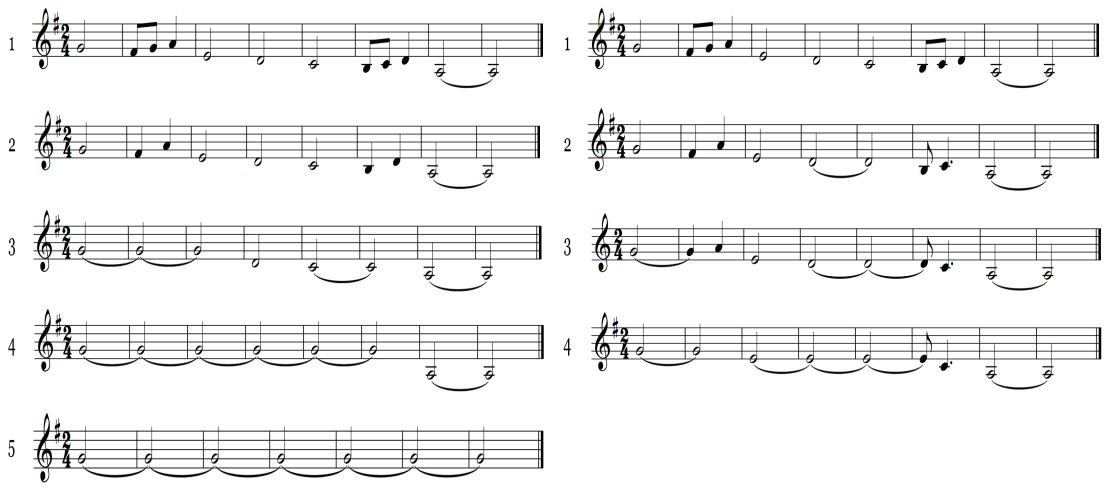


Figure 6.7.: Column (a) represents the prolongational reduction direct from the GTTM dataset (Hamanaka, Hirata, and Tojo 2007b), while column (b) represents the most probable parse tree from the PCFG trained with the second fold configuration of the prolongational reduction data, as seen in Figure 6.4. The top line on both sides is the original melody of *Pomp and Circumstance* by Sir Edward Elgar. Each row represents another level of depth removed from the trees, which creates a reduction of the previous level.

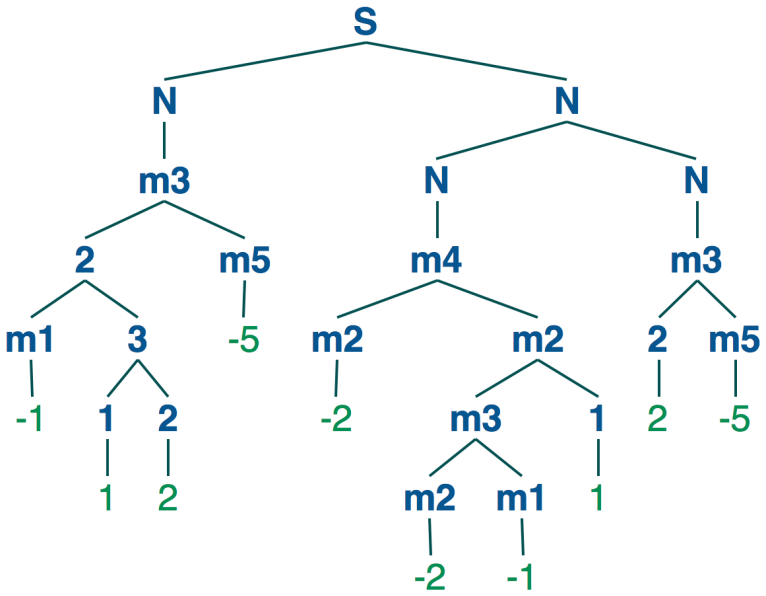


Figure 6.8.: The parse tree created by training the PCFG on the time-span reduction data, for the same melody shown in Figure 6.7: *Pomp and Circumstance* by Sir Edward Elgar.

its repetitions. Measures 2–3, for example, arpeggiate the underlying harmony rather clearly in each measure. However, these measures are not reduced at all in the second and third rows of Figure 6.9a. This is again due to the lack of production rules that describe the interval movements of melodies when they are arpeggiating a chord. Figure 6.10 shows the parse tree for *Für Elise*. This parse tree is also a good example of how unbalanced a parse tree can be.

6.3. Discussion

Certainly, the results are not ideal, with only around a 15% efficacy, and not much of a difference between the performance of the PCFG when the model is trained on the two different types of training data. Perhaps the time-span tree data creates a slightly more effective PCFG particularly because the reduction decisions are based more on meter; the embellishment rules defined in the PCFG seems to apply best to surface melodic figures, which rely on metrical placement. As discussed, there are limitations in the design and implementation of the underlying CFG, as was noted with the "New" rule, and the lack of metrical information in the data encoding. When only a certain percentage of nodes are even eligible to have an accurate result (all those that do not use the "New" rule), there is certainly a limitation in the design. When reducing a melody to a deeper level, the inaccuracies in the estimation of the surface-level reductions quickly compound.

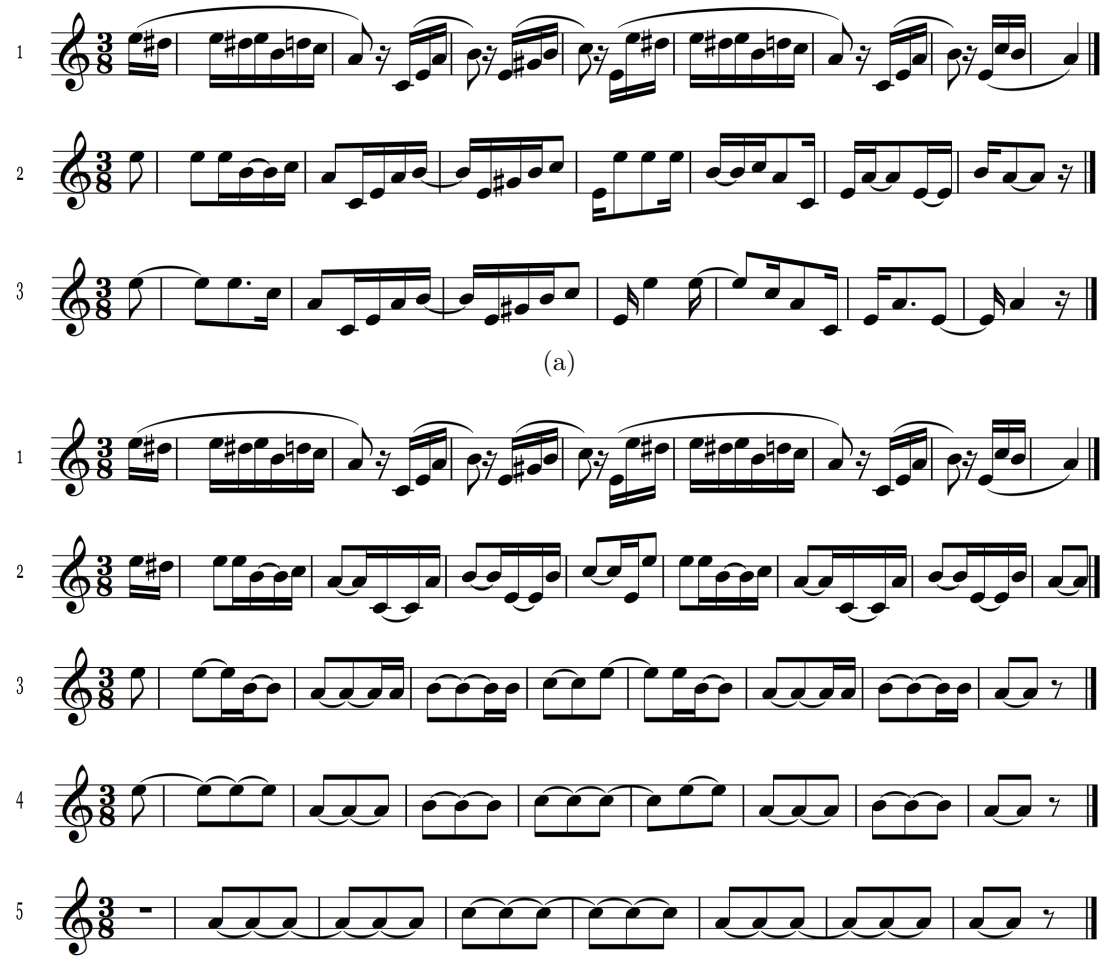


Figure 6.9.: An example of the melodic reductions produced by a time-span parse tree (a) and a time-span GTTM tree. The first line of both blocks is the original melody of *Für Elise* by Ludwig van Beethoven. Each row represents another level of depth removed from the trees, which creates a reduction of the previous level.

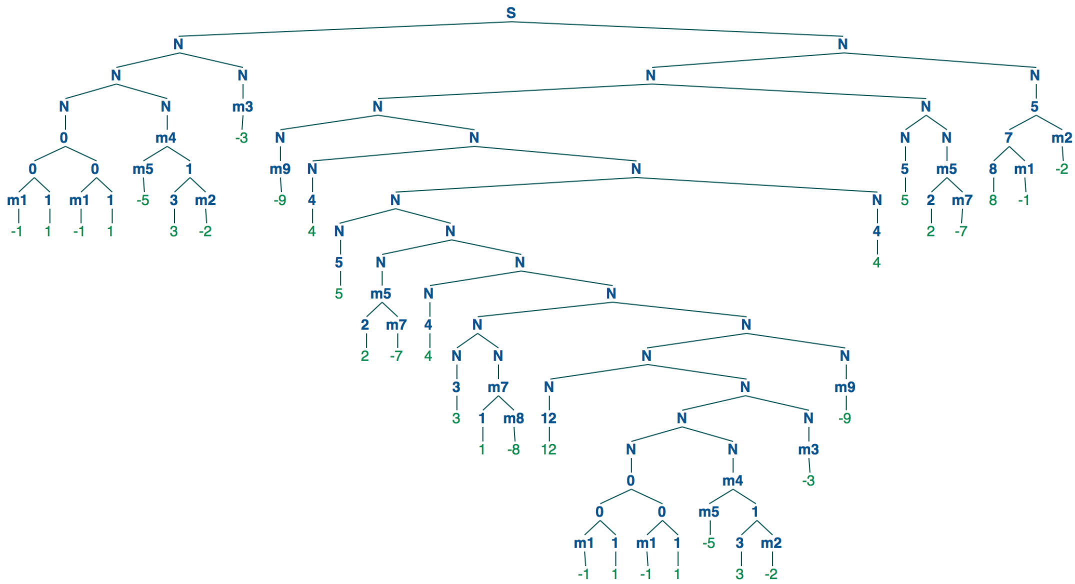


Figure 6.10.: The parse tree created by training the PCFG on the time-span GTTM data, for the melody *Für Elise* by Ludwig van Beethoven.

Some of the features of the PCFG, however, work quite well for melodic reduction. In Figure 6.9a, for example, identical passages were reduced in an identical way. This is due to the probabilistic nature of the PCFG. Similarly, the reduction of those repeated passages were almost perfect even at the second level of reduction, as compared with the solution reductions. Furthermore, barring the known limitations, the melodic embellishments created closer to the melodic surface are usually at least a viable solution. It seems that if those limitations were overcome, the performance could improve quite a lot. Some potential solutions to the given problems, and other future work will be the topic of the next chapter.

7. Conclusion

This thesis investigated the efficacy of a Probabilistic Context-Free Grammar (PCFG) when designed to automatically reduce melodies. The evaluation was done using a dataset of melodic trees, performing cross-validation after employing supervised learning to train the model. Previously, the PCFG had not been tested on a database of ground truth annotations, such as the one utilizing The Generative Theory of Tonal Music (GTTM) (Hamanaka, Hirata, and Tojo 2007b). Chapter 1 introduced the concept of melodic reduction, and some of its relevant uses. Chapter 2 gave a more detailed background of the critical techniques of Natural Language Processing (NLP) and the reductive theory behind the annotations that were used. Chapter 3 presented a history of research for not only melodic reduction, but also the relevant research for tasks related to melodic reduction, such as melodic segmentation and melodic parallelism. Chapter 4 explained the process of creating a PCFG, by first designing a Context-Free Grammar (CFG) and then utilizing the GTTM database created by Hamanaka et al. (2007b) to calculate the probability distributions for each rule of the PCFG. It also discussed the difficulties in translating the annotation format into the necessary input format for supervised training of a PCFG. Chapter 5 described the evaluation method, namely the comparison of tree structures.

Chapter 6 described the experiment and presented the results. Both the time-span tree and the prolongational tree data were used for the training of the PCFG. The most probable tree was parsed for each melody in the given test set, and those parse trees were compared with the ground truth GTTM trees. The evaluation method for each result aligned the leaves from the two trees, and performed a bottom-up tree comparison. It was found that the method had some critical limitations. Namely, the inability to describe every interval movement with the melodic embellishment rules eliminated the possibility of accurately estimating the higher levels of the tree structures. The lack of a proper representation of harmony and meter in the data encoding was a limiting factor. However, given the limitations, the PCFG was effective for the more surface-level melodic figures as was seen in Figure 6.9. If the limitations in the design were overcome, the PCFG for melodic reduction would certainly have a better performance.

7.1. Summary of Contributions

The main contribution of this work is the systematic evaluation of a PCFG for melodic reduction, using an existing database of expert annotations. Previously, the use of a PCFG for melodic reduction had been done, but supervised learning had not been performed with this model, and the results were not evaluated against a test set of ground truth melodic reductions.

Secondly, this thesis presents a unique solution for automatically generating the rules of a CFG using mathematical constraints. Using a PCFG for the processing and estimation of integer intervals can create a large combination of rules, which would be unwieldy to notate by hand. The framework developed allows for the specification of integer ranges for each of the right-hand side values, as well as the ability to define constraints (in the form of logical conditions) for validating when the current combination of right-hand side values constitutes a proper rule.

7.2. Future Work

There is much room for improvement for the current experiment. Namely, all of the following features would likely improve the model of the underlying system:

- Add rules to the CFG that allows the ruleset to describe every possible interval sequence without using the "New" Rule. This section will specifically explore the creation of harmonic grammar rules.
- Augmenting the PCFG model by changing the method of comparison for grammar rules. Ideally, the embellishment rule probabilities should be lumped together for each rule, even if the individual intervals for each left-hand side are different.
- Incorporate rhythm and/or meter into the representation.

This thesis will explore each of these possible improvements in this chapter.

Similarly, this thesis will briefly explore the potential uses of the results of the experiments. One feature of probabilistic systems is that the resulting probabilistic distributions can be sampled to generate instances of the modelled sequences. A trained PCFG for melodic reduction can be used to instead generate and embellish melodies. This process is also explored in this chapter.

7.3. Adding Harmony

The change that would be most fundamental to the improvement of the algorithm is to somehow ensure that every possible intervallic movement is covered by integer-based grammar rules. This change would ensure that every intervallic movement is described as part of some musical rule. As was seen in the example reduction for the song Für Elise in Figure 6.9, the melodic reduction fails to reduce any of the notes of the arpeggios in the melody. The solution reduction, on the other hand, reduces the melody in all measures, including those with arpeggios. Because the arpeggios cannot be reduced by the PCFG, the "New" rule must be used, making it impossible to continue the reduction on a higher level. Therefore, one large step in the right direction towards improving the reductional capabilities of the model would be to incorporate harmonic rules into the CFG. Currently, harmony is ignored. Not only does this leave many intervals unaccounted for in the ruleset that could be accounted to harmonic arpeggiation, but it also excludes the embellishment rules that require harmony from the ruleset.

The difficulty in creating a series of rules that describe the harmonic context of a piece is that the current representation encodes the relative pitch distance between two notes, as opposed to the set membership of a note; whether or not a note is part of a triad can be considered a type of set membership. However, it would not be impossible to represent a triad using relative pitch distances. As a demonstration of the potential for encoding harmonic context in a CFG, a method for encoding an ascending triad arpeggiation in a CFG is presented.

A triad has three possible notes, and can span any octave. Since every rule must represent relative pitch distance, it would be possible to compute the relative pitch between every note combination of the three notes. To simplify matters, a range of only one octave is considered for each individual interval. For the purpose of this experiment, a minor triad is modelled, which consists of a root note, a minor third above the root, and a perfect fifth above the root. In pitch-class terms, the set consists of a root note and the intervals of both three and seven above that root note. In order to truly create a triad, all of the notes within that triad must be visited. To further simplify the issue, the creating harmony grammar will only consider arpeggios that move up in pitch. To begin, a rule is created that represents the transition from the root note to the third of the the triad, followed by the transition from the third to the fifth of the triad:

ROOT → 3 4

This rule completes the triad, by visiting each note in the triad, starting from whatever note was first played. This by no means covers all the possible triads. Therefore, an abstraction is made to represent moving to the third of the triad. Then a rule is created for the third of the triad moving to the fifth, which will complete the triad:

ROOT → 3 THIRD
THIRD → 4

A difficulty occurs when the abstraction is made to create rules starting at each of the triad's representative notes (e.g. `ROOT`, `THIRD`, `FIFTH`). It would be ideal to represent the case for any time that note is visited. If a new rule were created for the fifth of the triad, the weakness of the abstraction is illuminated:

ROOT → 3 THIRD
THIRD → 4
FIFTH → 8 THIRD

The `FIFTH` rule now also utilizes the `THIRD` rule. If both the `FIFTH` and `THIRD` rules are used in succession, however, the resulting sequence would be a complete grammatical sentence, but a triad would not be formed. Instead, an interval of eight is followed by an interval of four—the notes have moved from the fifth of the triad, to the third, and back to the fifth over the span of an octave. It is therefore necessary to codify which triad notes have been visited before landing on a particular note, as follows:

S → ROOT | THIRD | FIFTH
ROOT → 3 ROOT_THIRD | 7 ROOT_FIFTH

THIRD → 4 THIRD_FIFTH | 9 THIRD_ROOT
FIFTH → 5 FIFTH_ROOT | 8 FIFTH_THIRD
ROOT_THIRD → 4
ROOT_FIFTH → 8
THIRD_FIFTH → 5
THIRD_ROOT → 7
FIFTH_ROOT → 3
FIFTH_THIRD → 8

The left-hand side values that represent two notes (e.g., `THIRD_ROOT`) provide a type of memory that allows the grammar to ensure that certain notes have been visited. In its current state, the grammar supports any combination of three ascending notes that will lead to the completion of the triad. It would be ideal to allow repetition of notes that are in the triad as well. In order to accomplish this, another memory state is added that represents the triad being completed already. These states are labelled with an `X` to represent that any interval that leads to a note in the triad is acceptable.

S → ROOT | THIRD | FIFTH
ROOT → 3 ROOT_THIRD | 7 ROOT_FIFTH
THIRD → 4 THIRD_FIFTH | 9 THIRD_ROOT
FIFTH → 5 FIFTH_ROOT | 8 FIFTH_THIRD
ROOT_THIRD → 4 | 4 FIFTHX
ROOT_FIFTH → 8 | 8 THIRDX
THIRD_FIFTH → 5 | 5 ROOTX
THIRD_ROOT → 7 | 7 FIFTHX
FIFTH_ROOT → 3 | 3 THIRDX
FIFTH_THIRD → 8 | 8 ROOTX
ROOTX → 3 | 7 | 3 THIRDX | 7 FIFTHX
THIRDX → 4 | 9 | 4 FIFTHX | 9 ROOTX
FIFTHX → 5 | 8 | 5 ROOTX | 8 THIRDX

From this CFG, there is still one situation that is not covered. For the rules that represent having visited two notes (e.g., `ROOT_THIRD` or `FIFTH_ROOT`), it is possible to have an interval that moves to another two-note rule. For example, the `ROOT_THIRD` represents being on the third note of the triad while having visited the root of the triad. If a major sixth interval occurs next, the triad will not be completed—rather, it would represent being back on the root note and having visited both the root and the third of the triad. In this case, it would return to the `THIRD_ROOT` rule. With the final addition of the situation described (for all two-note rules), the resulting CFG is:

S → ROOT | THIRD | FIFTH
ROOT → 3 ROOT_THIRD | 7 ROOT_FIFTH
THIRD → 4 THIRD_FIFTH | 9 THIRD_ROOT

FIFTH → 5 FIFTH_ROOT | 8 FIFTH_THIRD
ROOT_THIRD → 4 | 4 FIFTHX | 9 THIRD_ROOT
ROOT_FIFTH → 8 | 8 THIRDX | 5 FIFTH_ROOT
THIRD_FIFTH → 5 | 5 ROOTX | 8 FIFTH_THIRD
THIRD_ROOT → 7 | 7 FIFTHX | 3 ROOT_THIRD
FIFTH_ROOT → 3 | 3 THIRDX | 7 ROOT_FIFTH
FIFTH_THIRD → 9 | 9 ROOTX | 4 THIRD_FIFTH
ROOTX → 3 | 7 | 3 THIRDX | 7 FIFTHX
THIRDX → 4 | 9 | 4 FIFTHX | 9 ROOTX
FIFTHX → 5 | 8 | 5 ROOTX | 8 THIRDX

This final grammar generates every possible interval combination for ascending arpeggios of a minor triad. The triad can be rooted on any note, and can start from any note within the triad. The Natural Language Toolkit (NLTK) package provides a module for randomly generating sentences of the language defined by a CFG. This module was used to generate the following example sequences in Table 7.1 from the triadic grammar just described. The file that contains the CFG sampling code is *generate_arpeggio.py*. For clarity, each right-hand side option of the start rule (`ROOT`, `THIRD`, and `FIFTH`) was notated, and the interval sequences were used to generate the corresponding notes of the C minor triad. Each column starts from the respective note from the C minor triad.

	ROOT		THIRD		FIFTH	
D.	Intervals	Notes	Intervals	Notes	Intervals	Notes
4	[3, 4]	C, Eb, G	[4, 5]	Eb, G, C	[5, 3]	G, C, Eb
5	[3, 4, 5]	C, Eb, G, C	[4, 5, 3]	Eb, G, C, Eb	[5, 3, 4]	G, C, Eb, G
5	[3, 4, 8]	C, Eb, G, Eb	[4, 5, 7]	Eb, G, C, G	[5, 3, 9]	G, C, Eb, C
5	[3, 9, 7]	C, Eb, C, G	[4, 8, 9]	Eb, G, Eb, C	[5, 7, 8]	G, C, G, Eb
4	[7, 8]	C, G, Eb	[9, 7]	Eb, C, G	[8, 9]	G, Eb, C
5	[7, 8, 4]	C, G, Eb, G	[9, 7, 5]	Eb, C, G, C	[8, 9, 3]	G, Eb, C, Eb
5	[7, 8, 9]	C, G, Eb, C	[9, 7, 8]	Eb, C, G, Eb	[8, 9, 7]	G, Eb, C, G
5	[7, 5, 3]	C, G, C, Eb	[9, 3, 4]	Eb, C, Eb, G	[8, 4, 5]	G, Eb, G, C
10	[7, 8, 9, 7, 5, 3, 9, 7]	C, G, Eb, C, G, C, Eb, C, G	[9, 3, 9, 7, 8, 4, 8, 9]	Eb, C, Eb, C, G, Eb, G, Eb, C	[8, 4, 8, 4, 5, 3, 4, 8]	G, Eb, G, Eb, G, C, Eb, G, Eb

Table 7.1.: Ascending arpeggios generated from the CFG for a minor triad. The "D." column stands for grammar tree "Depth", and is proportional to the number of intervals contained in each grammar string. The grammar was run with the three different starting rules of `ROOT`, `THIRD`, and `FIFTH`. The intervals were then converted to individual notes starting from the root, third, and fifth of the C minor triad, respectively. All trees of depth five or less are included, as well as one randomly-selected grammar string of depth ten.

This CFG for a minor triad could be modified easily to create a grammar for major

triads, or diminished triads. It could also be modified to support chords with a variable number of member notes, like two or four. Could it also be used to model a sequence of varying chords? The difficulty with extending this grammar to model sequences of chords is in the transitions from chord to chord. Suppose the underlying harmony was a C minor triad followed by a F major triad. In order to create rules that modelled both triads as well as the transition between them, one would have to know exactly how each note from the first triad can transition to each of the notes of the second triad. For example, if the transition occurred from the third of the first triad to the root of the second triad, it would be an interval of two semitones from Eb to F, whereas if the starting note were the root of the first triad, the interval would be five semitones from C to F. The requirement of describing the situation in terms of relative intervals in this case seems to constrict the grammar from being applied. The main difficulty is that, with the minor triad grammar, the interval transition from that minor triad to another triadic grammar depends very much on the final note that occurs in the first triad. This is defined in the grammar by which rule has happened last. In a Context-*Free* Grammar, one cannot create a transitional rule that takes into account the state of the previous rule. For this reason, a melody that contains an underlying sequence of changing chords cannot be accurately modelled with the above grammar representation of harmony (it would be able to model a melody that only has one underlying chord). Likely, to incorporate harmony into a model with changing harmonic content, one would need a more general abstraction of harmony.

If a melody remains in a single harmonic context (without transitioning to a new underlying chord) it would be possible to extend the arpeggiating grammar above with the rules of embellishment. Instead of simply terminating on each interval integer, one could add the non-terminal rules that were used in the experiment to extend the possibilities to notes outside of the current harmonic context.

7.4. Augmenting the Model

There is a limitation of the chosen model in regards to the representation of the rules. The limitation is that some of the embellishment rules have multiple right-hand side rules that describe the same process applied to different interval spans. This creates a different left-hand side value for each of the rule applications. Similarly, each rule must be resolved to the string-based representation of intervals; instead of a conditional comparison of numerals, each integer interval must be compared with its string representation. This means that the embellishment rules are not considered for any given interval based on their overall frequency of occurrence, rather they are considered based on their frequency of occurrence *at that interval*. There could be a situation, for example, where the Escape Tone Rule happens in the training set frequently on a minor chord, where it embellishes the minor third. This information would not then be applicable to an augmented chord, for example, because an augmented chord consists of two consecutive major thirds.

Below is a further example with the Escape Tone Rule. With the current string-based representation, one must create an individual rule for each interval size that represents

a leap. As shown in Section 4.2, the different leap sizes necessitate the creation of one hundred and sixty-eight rules. Each of these rules create their own probability distribution, and therefore the probability of an Escape Tone happening is not properly represented. In fact, some of the production rules that describe the Escape Tone Rule will actually collide with other embellishment rules. Take the Escape Tone Rule applied to an interval of size four:

$4 \rightarrow 5$ m1
$4 \rightarrow 6$ m2

The challenge here is not in that these rules are incorrect, but they resolve to a left-hand side that is not unique to the embellishment rule that is being modeled. Here are the other rules in the CFG that have a left-hand side of the interval four:

$4 \rightarrow 2\ 2$
$4 \rightarrow$ '4'

Contained in the CFG is also a production rule that represents the Passing Tone Rule applied over an interval of four semitones. Ideally, the probabilities should all be grouped by the type of embellishment rule even if the size of the intervals are different. If, instead of strings, the CFG rule actually contained in its implementation the ability to apply mathematical constraints (such as those that were made when constructing the CFG) *at the time of parsing an input sequence*, then it would be possible to directly model the probability distribution of the rule, without distributing the probabilities over multiple right-hand side rules that represent the manifestation of that embellishment rule only at that particular interval. This would require the creation of a new parsing algorithm, where every string comparison is instead replaced with a mathematical constraint validator. The result would be a constraint-based CFG. This would improve the experiment because the probability for each rule would not depend on the interval at which the rule is being applied, and would more closely represent the frequency with which the rule occurred in the training set.

7.5. Representing Rhythm and Meter

An additional change that would benefit the model would be the addition of rhythm and/or meter into the data representation. As was shown in the reduction examples for the melody *Pomp and Circumstance* in Figure 6.7, the lack of metrical information allows for the application of the embellishment rules in unusual places. Specifically the Escape Tone Rule is applied to the last three notes, which utilizes a very weak metrical position for the first parent note, which has the metrical position of the second beat of a measure in time signature $\frac{3}{4}$.

There are different options for encoding the metrical positions of notes. A simple method is to couple the intervals with an interval of the metric level of each note. Gilbert and Conklin (2007) used this approach using what they called a "metric delta". To compute this metric delta, the metric levels were grouped in a hierarchy. The hierarchical

levels represented the significance of each of the beats within a measure. The first level is the downbeat of the first beat of the measure, the second level is the downbeat of the third beat, then the down beats of the second and fourth beat were grouped as a level, then the upbeats, then notes that fell on the sixteenth notes. Similar to the intervals between the notes' pitches, an interval between these metric levels was computed for each data point, and the pitch interval and metric interval were grouped as a single observation. This approach again expands the probability distributions and categorizes the same embellishment rule into even more production rules. One disadvantage of this is that it creates more sparsity in the resulting probabilistic distributions, and also creates a bigger disparity between the musical embellishment rules, and their implementation in the PCFG; each embellishment rule would have a larger multiple of different production rules, spreading the probabilities over an even wider set of left-hand side possibilities. However, it would have the advantage of the ability to avoid the use of certain notes with poor metrical positions as parent notes of embellishment rules, and could possibly prevent the mistakes like the one shown in the *Pomp and Circumstance* example from occurring.

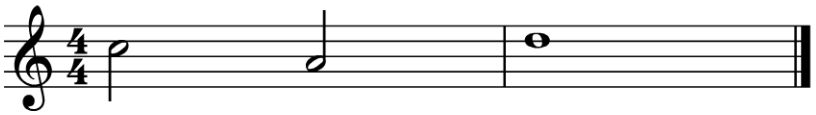
7.6. Melodic Generation Examples

The goal of the following demonstration is not to assert that the PCFG created is an effective tool for composition; the melodies created below do not compare to the creations of expert composers, by any means. Rather, the aim is to show the possibilities for applying a trained grammar. By definition, any sequence that is generated by sampling a probabilistic grammar is part of the *language* that the grammar defines. Therefore, if a particular style of melodies is properly modelled, the output sequences generated by that model should also be recognizable as a manifestation of that style. With these particular results it is evident that is not the case. However, given a PCFG that passes a higher-level evaluation, one could utilize it to potentially correct existing melodies (as a sort of musical "spell-check"), or to automatically compose melodies of moderate quality in a large volume.

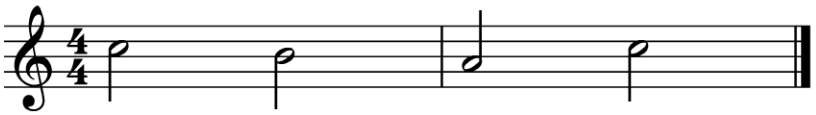
7.6.1. Sampling the PCFG

With a trained PCFG like the one created in this experiment, it is possible to generate output sequences by sampling the probabilities for each rule. Each set of rules for a given left-hand side has a probability for each right-hand side possibility. Each rule's right-hand side probabilities sum to one. By using the probability distributions, one can create a new sequence that is representative of the grammar, through the process of random sampling.

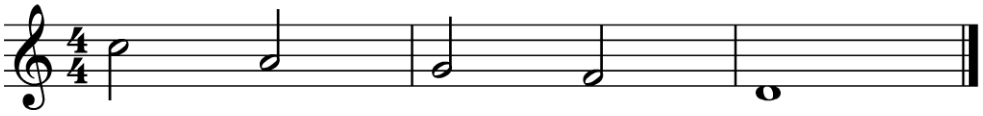
The NLTK provides a function for each probabilistic rule of a PCFG that generates example sequences for a PCFG. To sample a distribution, the process is simple. One would need to assign a mathematical range to each right-hand side based on the proportion of its probability. Then, a random number must be generated, and the range that contains that random number is the newly-sampled rule. The NLTK toolkit



(a) A generated melody with the interval sequence of `[-3, 5]`.



(b) A generated melody with the interval sequence of `[-1, -2, 3]`.



(c) A generated melody with the interval sequence of `[-3, -2, -2.  -3]`.

Figure 7.1.: Melodies generated by random sampling of a PCFG trained on the fifth fold of the prolongational reduction tree dataset.

was used for this process as well. It is possible to use this feature to generate a whole tree structure, based on recursively sampling the right-hand side of each rule generated, starting with the "Start" rule. The file that contains the sampling code for the PCFG is *generate_arpeggio.py*, part of the PCFGMelodicReduction repository.

An example of a generated melody is shown in Figure 7.1a. The onset for each note is arbitrarily assigned to intervals of two beats in order to avoid emphasizing individual notes rhythmically. Similarly, each generated melody was chosen to start on the note C. For the purposes of the following examples, the trained grammar used for the melodic generation was created from the fifth fold of the prolongational reduction trees. Through the generation process, the sampled grammar creates a sequence of pitch intervals, which are computed in succession from the starting note. The melody in Figure 7.1a happens to stay within the C major scale, but because there is no concept of scale or harmony encoded in the grammar, the notes are able to move with any interval within the defined range—even those outside of the C major scale. The generated interval sequences will vary in length as well. The only rule that will add extra intervals is the "New" Rule: `N` $\rightarrow$ `N N`. It also has a right-hand side production for every possible integer terminal as well as all the embellishment rule non-terminals. The right-hand side that does extend the melody occurs with an average probability of 16.68% over the five folds. When this rule is applied recursively, it soon samples the other possible right-hand side productions, which are all terminals or embellishment rules (which lead to terminals), thus ending the melody. Two additional generated melodies are shown in Figure 7.1b and Figure 7.1c. The fact that these melodies also remain in the C major scale is coincidence.

Melodic Embellishment With Selective Sampling

It is also possible to use sampling in order to utilize the trained PCFG for particular purposes. For the situation in which a melody already exists, it is possible to restrict the set of grammar rules that are considered when sampling in order add notes to the existing melody. Another algorithm was created in order to selectively sample only the embellishment rules. The process is described below:

- Parse the melody using the probabilistic grammar, in order to create a parse tree
- For every leaf in the tree
  - Select the parent non-terminal of the given leaf
  - Find all right-hand side productions of the parent non-terminal
  - Select all right-hand side productions that are of length greater than one
  - Normalize the aggregated probabilities of the new right-hand side rule subset
  - Sample from the new probabilistic distribution that includes only those rules that will embellish the given interval
    - Find the corresponding terminal to each of the new child non-terminals created from sampled embellishment rule
  - Replace the leaf with the new grammar tree created from the embellishment rule and its child non-terminals

Some of these steps warrant further explanation. The process will be detailed for the first generated melody from Figure 7.1a. The first step is to parse the melody to create the parse tree. Applying the PCFG to this melody yields the following tree. In this case, each set of parentheses represent the application of a new non-terminal:

```
(S (N (m3 -3)) (N (5 5)))
```

The series of rules that created this tree are:

$\{$`S` $\rightarrow$ `N, N` $\rightarrow$ `N N, N` $\rightarrow$ `m3, m3` $\rightarrow$ `'-3', N` $\rightarrow$ `5, 5` $\rightarrow$ `'5'`$\}$.

The tree graph is shown in Figure 7.2.

The set of leaves in the tree depicted in Figure 7.2 are simply `'-3'` and `'5'`. Iterating through each of these, `'-3'` is selected first, and its parent non-terminal is identified as simply `m3`. The next step involves finding all the right-hand side production rules for the non-terminal `m3`, which amount to the following set of rules. The full grammar rule is provided for each, as well as the probability with which that rule occurred in the

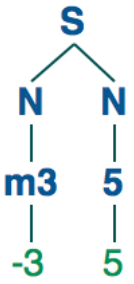


Figure 7.2.: The parse tree from the generated melody in Figure 7.1a.

training set, for every time that non-terminal was visited:

m3 → m1 m2	[0.0763547]
m3 → m2 m1	[0.0738916]
m3 → 2 m5	[0.00985222]
m3 → m5 2	[0.00738916]
m3 → '-3'	[0.832512]

From this list of productions, it is evident that three separate embellishment rules can be applied of the interval spanning three semitones: the Passing Tone Rule, the Cambiata, and the Escape Tone Rule. The Passing Tone can be applied for a sequence of negative two semitones followed by an interval of negative one semitone, or vice versa. For this interval, the Cambiata has only one right-hand side possibility: the negative five-semitone interval followed by an interval of two. The right-hand side rule that represents the other possible Cambiata at this interval (an interval of minus four semitones followed by an interval of one semitone) is missing. The reason it is not part of this set is that it was never observed in the training set. It received a zero probability, and was pruned from the PCFG. The same goes for the right-hand side rule representing both of the Escape Tone Rule options: the one-semitone interval followed by the negative four-semitone interval and the two-semitone interval followed by a negative five-semitone interval.

Following the steps for embellishment above, the rules are now pruned so that only the rules with a length of two or greater are left. Similarly, the probabilities are re-normalized so that in the new subset, the probabilities still accumulate to a total of 1.0. The new production subset would consist of these new rules:

m3 → m1 m2	[0.45588161539931]
m3 → m2 m1	[0.4411754871991]
m3 → 2 m5	[0.05882343809706]
m3 → m5 2	[0.04411754871991]

Sampling from this new probability distribution yields the `m3` → `m2 m1` rule. Each

of the values in the new right-hand side are also non-terminals, so they must be expanded. Instead of continuing to sample the production rules, which might yield more non-terminals, the process instead seeks the terminal that applies to each of the non-terminals. A small tree is created that represents the new embellishment: `(m2 '-2') (m1 '-1')`. The leaf node of `'-3'` can then be replaced by the new tree, finalizing the embellishment for this leaf node. The process would then be continued for the next leaf in the original tree, which is `'5'`. The probabilistic productions of the parent of this final leaf are as follows:

$5 \rightarrow$ '5' [0.974074]
$5 \rightarrow$ 7 m2 [0.0111111]
$5 \rightarrow$ m1 6 [0.00740741]
$5 \rightarrow$ m2 7 [0.0037037]
$5 \rightarrow$ 6 m1 [0.0037037]

After pruning and re-normalizing, the production set becomes the following:

$5 \rightarrow$ 7 m2 [0.42856977551493]
$5 \rightarrow$ m1 6 [0.2857135693898]
$5 \rightarrow$ m2 7 [0.14285659183831]
$5 \rightarrow$ 6 m1 [0.14285659183831]

For this rule, the `7 m2` rule was randomly selected via sampling of the embellishment rule subset. The embellishments are then inserted back into the original tree. The final embellished tree can be found in Figure 7.3.

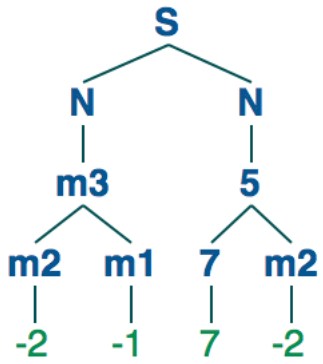
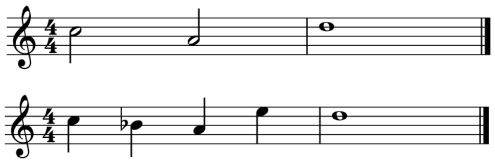
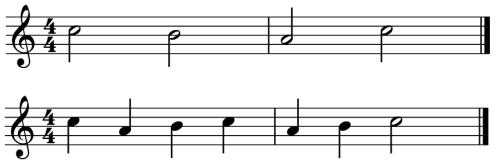


Figure 7.3.: The generated melody in Figure 7.1a after being embellished.

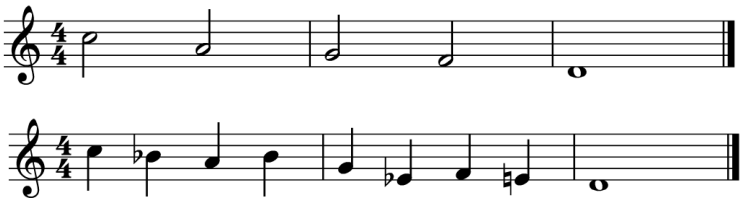
The process of embellishment with selective sampling was performed on all three of the previously generated melodies. The resulting embellished melodies are shown along the original melodies in Figure 7.4.



(a) A generated melody with the interval sequence of `[-3, 5]`, with its embellished counterpart of interval sequence `[-2, -1, 7, -2]`



(b) A generated melody with the interval sequence of `[-1, -2, 3]` above its embellished counterpart of interval sequence `[-3, 2, 1, -3, 2, 1]`



(c) A generated melody with the interval sequence of `[-3, -2, -2, -3]` above its embellished counterpart of interval sequence `[-2, -1, 1, -3, -4, 2, -1, -2]`

Figure 7.4.: Melodies generated by random sampling of a PCFG and then embellished with selective sampling. The PCFG was trained on the fifth fold of the prolongational reduction tree dataset.

7.7. Recapitulation

This thesis has shown that the PCFG is a dynamic tool, allowing for the hierarchical modelling of sequential data. With a growing number of databases available, techniques for the systematic evaluation of digital representations of musical scores are becoming more useful and relevant to the fields of music theory and musicology. Systematic evaluations provide not only a way to validate the potential in certain models, but also to hone and improve upon data representations of musical information when utilized for specific tasks. The level of intuition of expert musicologist or composer might never be attained by software tools, but certainly large-scale analysis of music provides a different perspective on musical challenges, and a larger platform for experts to apply their skills.

References

Abdallah, S. A., and N. E. Gold. 2014. Comparing models of symbolic music using probabilistic grammars and probabilistic programming. In *Proceedings of the International Computer Music Conference*, Athens, Greece, 1524–31.

Abdallah, S. A., N. E. Gold, and A. Marsden. 2016. Analysing symbolic music with probabilistic grammars. In D. Meredith (Ed.), *Computational Music Analysis*, 157–89. Cham, Switzerland: Springer International.

Backus, J. W. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the International Conference for Information Processing*, Paris, France, 125–31.

Baroni, M., R. Brunetti, L. Callegari, and C. Jacoboni. 1982. A grammar for melody: Relationships between melody and harmony. In M. Baroni and L. Callegari (Eds.), *Musical Grammars and Computer Analysis*, Florence, Italy, 201–18.

Baroni, M., and C. Jacobini. 1975. Analysis and generation of Bach's chorale melodies. In *Proceedings of the International Congress on the Semiotics of Music*, Belgrade, Yugoslavia, 125–34.

Baroni, M., and C. Jacoboni. 1978. *Proposal for a grammar of melody: The Bach Chorales*. Montreal, Canada: Les Presses de l'Université de Montréal.

Beach, D. 1969. A Schenker bibliography. *Journal of Music Theory* 13 (1): 2–37.

Bel, B., and J. Kippen. 1992. Modelling music with grammars: Formal language representation in the Bol processor. In A. Marsden and A. Pople (Eds.), *Computer Representations and Models in Music*, 207–38. London, United Kingdom: Academic Press.

Bernabeu, J. F., J. Calera-Rubio, and J. M. Iñesta. 2011. Classifying melodies using tree grammars. In *Proceedings of the Iberian Conference on Pattern Recognition and Image Analysis*, Las Palmas de Gran Canaria, Spain, 572–9.

Bod, R. 1998. *Beyond grammar: An experience-based theory of language*. Stanford, CA: CSLI Publications.

Bod, R. 2002. Memory-based models of melodic analysis: Challenging the Gestalt principles. *Journal of New Music Research* 31: 27–37.

Booth, T. L. 1969. Probabilistic representation of formal languages. In *Proceedings of the Symposium on Switching and Automata Theory*, Waterloo, Canada, 74–81.

Buxton, W., W. Reeves, R. Baecker, and L. Mezei. 1978. The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal* 2 (4): 10–20.

Cambouropoulos, E. 2001. The local boundary detection model (LBDM) and its application in the study of expressive timing. In *Proceedings of the International Computer Music Conference*, Havana, Cuba, 17–22.

Cambouropoulos, E. 2006. Musical parallelism and melodic segmentation: A computational approach. *Music Perception* 23: 249–68.

Cambouropoulos, E., M. Crochemore, C. Iliopoulos, M. Mohamed, and M. Sagot. 2005. A pattern extraction algorithm for abstract melodic representations that allow partial overlapping of intervallic categories. In *Proceedings of the International Conference on Music Information Retrieval*, London, United Kingdom, 167–74.

Chomsky, N. 1956. Three models for the description of language. *Institute of Radio Engineers Transactions on Information Theory* 2: 113–24.

Chomsky, N. 1959. On certain formal properties of grammars. *Information and Control* 2 (2): 137–67.

Cleary, J. G., W. J. Teahun, and I. H. Witten. 1995. Unbounded length contexts for PPM. In *Proceedings of the Data Compression Conference*, Snowbird, UT, 52–61.

Cocke, J. 1969. *Programming languages and their compilers: Preliminary notes*. New York, NY: Courant Institute of Mathematical Sciences, New York University.

Collins, M. J. 1999. Head-driven statistical models for natural language parsing. *Computational Linguistics* 29 (4): 589–637.

Cooper, G., and L. B. Meyer. 1960. *The rhythmic structure of music*. Chicago, IL: University of Chicago Press.

Cope, D. 1996. *Experiments in musical intelligence*, Volume 12. Madison, WI: AR Editions.

Crochemore, M. 1981. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters* 12 (5): 244–50.

de la Puente, A. O., R. S. Alfonso, and M. A. Moreno. 2002. Automatic composition of music by means of grammatical evolution. In *Proceedings of the Conference on APL: Array Processing Languages: Lore, Problems, and Applications*, New York, NY, 148–55.

Ferrand, M., P. Nelson, and G. Wiggins. 2003. Unsupervised learning of melodic segmentation: A memory-based approach. In *Proceedings of the European Society for the Cognitive Sciences of Music Conference*, Hannover, Germany, 141–4.

Forte, A., and S. E. Gilbert. 1982. *An introduction to Schenkerian analysis*. Cambridge, MA: W. W. Norton.

Frankland, B., and A. J. Cohen. 2004. Parsing of melody: Quantification and testing of the local grouping rules of Lerdahl and Jackendoff's "A generative theory of tonal music". *Music Perception* 21 (4): 499–543.

Friberg, A., R. Bresin, L. Frydén, and J. Sundberg. 1998. Musical punctuation on the microlevel: Automatic indentification and performance of small melodic units. *Journal of New Music Research* 27 (3): 271–92.

Gilbert, É., and D. Conklin. 2007. A probabilistic context-free grammar for melodic reduction. In *Proceedings for the International Workshop on Artificial Intelligence and Music, International Joint Conference on Artificial Intelligence*, Hyderabad, India, 83–94.

Good, M. 2001. MusicXML for notation and analysis. *The virtual score: representation, retrieval, restoration* 12: 113–24.

Granroth-Wilding, M. T. 2013. *Harmonic analysis of music using combinatory categorial grammar*. Ph.D. thesis, University of Edinburgh, Edinburgh, United Kingdom.

Hamanaka, M., K. Hirata, and S. Tojo. 2007a. FATTA: Full automatic time-span tree analyzer. In *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, 153–6.

Hamanaka, M., K. Hirata, and S. Tojo. 2007b. Implementing "A generative theory of tonal music". *Journal of New Music Research* 35 (4): 249–77.

Hoffmann, C. M., and M. J. O'Donnell. 1982. Pattern matching in trees. *Journal of the ACM* 29 (1): 68–95.

Jurafsky, D., and J. H. Martin. 2000. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition* (1st ed.). Upper Saddle River, NJ: Prentice Hall.

Kasami, T. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, Bedford, MA.

Kassler, M. 1963. A sketch of the use of formalized languages for the assertion of music. *Perspectives of New Music* 1 (2): 83–94.

Kay, M. 1986. Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck-Jones, and B. L. Webber (Eds.), *Readings in Natural Language Processing*, 35–70. San Francisco, CA: Morgan Kaufmann.

Kirlin, P. B. 2014. *A probabilistic model of hierarchical music analysis*. Ph.D. thesis, University of Massachusetts Amherst, Amherst, MA.

Kirlin, P. B., and D. D. Jensen. 2011. Probabilistic modelling of hierarchical music analysis. In *Proceedings of the International Conference of Music Information Retrieval*, Miami, FL, 393–8.

Koffka, K. 1935. *Principles of Gestalt psychology*. New York, NY: Harcourt, Brace & World.

Köhler, W. 1929. *Gestalt psychology*. New York, NY: Liveright.

Lartillot, O. 2010. Reflections towards a generative theory of musical parallelism. *Musicae Scientiae* 14 (1): 195–229.

Lerdahl, F., and R. Jackendoff. 1983. *A generative theory of tonal music*. Cambridge, MA: The MIT Press.

Lerdahl, F., and Y. Potard. 1986. *La composition assistée par ordinateur*. Rapports de recherche. Paris, France: Institut de Recherche et Coordination Acoustique/Musique, Centre Georges Pompidou.

Levenshtein, V. I. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10: 707–10.

Longuet-Higgins, H., and M. J. Steedman. 1971. On interpreting Bach. *Machine Intelligence* 6: 221–41.

Loper, E., and S. Bird. 2002. NLTK: The natural language toolkit. In *Proceedings of the Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, Volume 1, Stroudsburg, PA, 63–70.

MacKay, D. J. 2003. *Information theory, inference and learning algorithms*. Cambridge, United Kingdom: Cambridge University Press.

Marsden, A. 2005. Generative structural representation of tonal music. *Journal of New Music Research* 34 (4): 409–28.

Marsden, A. 2010, August 9-13. Recognition of variations using automatic Schenkerian reduction. In *Proceedings of the International Conference on Music Information Retrieval*, Utrecht, Netherlands, 501–6.

Mont-Reynaud, B., and M. Goldstein. 1985. On finding rhythmic patterns in musical lines. In *Proceedings of the International Computer Music Conference*, Burnaby, Canada, 391–7.

Nord, T. 1992. *Toward theoretical verification: Developing a computer model of Lerdahl and Jackendoff's generative theory of tonal music*. Ph.D. thesis, University of Wisconsin-Madison, Madison, Wisconsin.

Pasquier, N., Y. Bastide, R. Taouil, and L. Lakhal. 1999. Efficient mining of association rules using closed itemset lattices. *Information Systems* 24: 25–46.

Pearce, M. T., D. Müllensiefen, and G. A. Wiggins. 2010. Melodic grouping in music information retrieval: New methods and applications. In Z. Ra and A. Wieczorkowska (Eds.), *Advances in Music Information Retrieval*, Volume 274 of *Studies in Computational Intelligence*, 364–88. Berlin, Germany: Springer.

Pope, S. T. 1991a. A tool for manipulating expressive and structural hierarchies in music. In *Proceedings of the International Computer Music Conference*, Montreal, Canada, 324–7.

Pope, S. T. 1991b. *The well-tempered object: Musical applications of object-oriented software technology*. Cambridge, MA: MIT Press.

Rizo, D. 2010. *Symbolic music comparison with tree data structures*. Ph.D. thesis, University of Alicante, Alicante, Spain.

Roads, C. 1977. Composing grammars. In *Proceedings of the International Computer Music Conference*, San Diego, CA, 54–132.

Roads, C., and P. Wieneke. 1979, March. Grammars as representations for music. *Computer Music Journal* 3 (1): 48–55.

Ruwet, N. 1972. *Language, musique, poésie*. Paris, France: Éditions du Seuil.

Ruwet, N. 1975. Theorie et methodes dans les etudes musicales. *Musique en Jeu* 17: 11–36.

Sadie, S., and G. Grove. 1980. *The new Grove dictionary of music and musicians*. London, United Kingdom: Macmillan.

Schaffrath, H. 1995. The Essen folksong collection in the Humdrum kern format. Menlo Park, CA: Center for Computer Assisted Research in the Humanities.

Seneff, S. 1992. TINA: A natural language system for spoken language applications. *Computational Linguistics* 18 (1): 61–86.

Smoliar, S. W. 1976. Music programs: An approach to music theory through computational linguistics. *Journal of Music Theory* 20 (1): 105–31.

Smoliar, S. W. 1986. Reviewed work: Musical grammars and computer analysis. *Journal of Music Theory* 30 (1): 130–41.

Steedman, M. J. 1977. The perception of musical rhythm and metre. *Perception* 6 (5): 555–69.

Temperley, D. 2001. *The cognition of basic musical structures*. Cambridge, MA: The MIT Press.

Tenney, J., and L. Polansky. 1980. Temporal Gestalt perception in music. *Journal of Music Theory* 24 (2): 205–41.

Wertheimer, M. 1938. Laws of organization in perceptual forms. In W. D. Ellis (Ed.), *A Source Book of Gestalt Psychology*, 71–88. London, United Kingdom: Routledge & Kegan Paul.

Younger, D. H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control* 10 (2): 189–208.

Appendices

A. The Preference Rules of GTTM

Each of the following rules come from The Generative Theory of Tonal Music (GTTM) (Lerdahl and Jackendoff 1983). The page numbers are referenced after each rule.

Grouping Preference Rule (GPR) 1 Strongly avoid groups containing a single event. [43]

GPR 2 (Proximity) Consider a sequence of four notes $n_1 n_2 n_3 n_4$. All else being equal, the transition n_2–n_3 may be heard as a group boundary if

a. (Slur/Rest) the interval of time from the end of n_2 to the beginning of n_3 is greater than that from the end of n_1 to the beginning of n_2 and that from the end of n_3 to the beginning of n_4, or if

b. (Attack-Point) the interval of time between the attack points of n_2 and n_3 is greater than that between the attack points of n_1 and n_2 and that between the attack points of n_3 and n_4. [45]

GPR 3 (Change) Consider a sequence of four notes $n_1 n_2 n_3 n_4$. All else being equal, the transition n_2–n_3 may be heard as a group boundary if

a. (Register) the transition n_2–n_3 involves a greater intervallic distance than both n_1–n_2 and n_3–n_4, or if

b. (Dynamics) the transition n_2–n_3 involves a change in dynamics and n_1–n_2 and n_3–n_4 do not, or if

c. (Articulation) the transition n_2–n_3 involves a change in articulation and n_1–n_2 and n_3–n_4 do not, or if

d. (Length) n_2 and n_3 are of different lengths and both pairs n_1,n_2 and n_3,n_4 do not differ in length.

(One might add further cases to deal with such things as change in timbre or instrumentation.) [46]

GPR 4 (Intensification) Where the effects picked out by GPRs 2 and 3 are relatively more pronounced, a larger-level group boundary may be placed. [49]

GPR 5 (Symmetry) Prefer grouping analyses that most closely approach the ideal subdivision of groups into two parts of equal length. [49]

GPR 6 (Parallelism) Where two or more segments of the music can be construed as parallel, the preferably form parallel parts of groups. [51]

GPR 7 (Time-span and Prolongational Stability) Prefer a grouping structure that results in more stable time-span and/or prolongational reductions. [52]

Metrical Preference Rule (MPR) 1 (Parallelism) Where two or more groups or parts of groups can be construed as parallel, they preferably receive parallel metrical structure. [75]

MPR 2 (Strong Beat Early) Weakly prefer a metrical structure in which the strongest beat in a group appears relatively early in the group. [76]

MPR 3 (Event) Prefer a metrical structure in which beats of level L_i that coincide with the inception of pitch-events are strong beats of L_i. [76]

MPR 4 (Stress) Prefer a metrical structure in which beats of Level L_i that are stressed are strong beats of L_i. [79]

MPR 5 (Length) Prefer a metrical structure in which a relatively strong beat occurs at the inception of either:

a. a relatively long pitch-event,

b. a relatively long duration of a dynamic,

c. a relatively long slur,

d. a relatively long pattern of articulation,

e. a relatively long duration of a pitch in the relevant levels of the time-span reduction, or

f. a relatively long duration of a harmony in the relevant levels of the time-span reduction (harmonic rhythm). [84]

MPR 6 (Bass) Prefer a metrically stable bass. [88]

MPR 7 (Cadence) Strongly prefer a metrical structure in which cadences are metrically stable; that is, strongly avoid violations of local preference rules within cadences. [88]

MPR 8 (Suspension) Strongly prefer a metrical structure in which a suspension is on a stronger beat than its resolution. [89]

MPR 9 (Time-span Interaction) Prefer a metrical analysis that minimizes conflict in the time-span reduction. [90]

Time-span Reduction Preference Rule (TSRPR) 1 (Metrical Position) Of the possible choices for head of a time-span T, prefer a choice that is in a relatively strong metrical position. [160]

TSRPR 2 (Local Harmony) Of the possible choices for head of a time-span, T, prefer the choice that is

a. relatively intrinsically consonant,

b. relatively closely related to the local tonic. [161]

TSRPR 3 (Registral Extremes) Of the possible choices for head of a time-span, T, weakly prefer a choice that has

a. a higher melodic pitch

b. a lower bass pitch [162]

TSRPR 4 (Parallelism) If two or more time-spans can be construed as motivically and/or rhythmically parallel, preferably assign them parallel heads. [164]

TSRPR 5 (Metrical Stability) In choosing the head of a time-span T, prefer a choice that results in more stable choice of metrical structure. [165]

TSRPR 6 (Prolongational Stability) In choosing the head of a time-span T, prefer a choice that results in more stable choice of prolongational reduction. [167]

TSRPR 7 (Cadential Retention) If the following conditions obtain in a time-span T, then label the progression as a cadence and strongly prefer to choose it as head:

i. There is an event or sequence of two events $(e_1)e_2$ forming the progression for a full, half, or deceptive cadence.

ii. The last element of this progression is at the end of T or is prolonged to the end of T.

iii. There is a larger group G containing T for which the progression can function as structural ending. [170]

TSRPR 8 (Structural Beginning) If, for a time-span T, there is a larger group G containing T for which the head of T can function as structural beginning, the prefer as head of T an event relatively close to the beginning of T (and hence to the beginning of G as well). [170]

TSRPR 9 In choosing the head of a piece, prefer the structural ending to the structural beginning. [174]

Prolongational Reduction Preference Rule (PRPR) 1 (Time-span Importance) In choosing the prolongationally most important event e_k of a prolongational region (e_i–e_j), strongly prefer a choice in which e_k is relatively time-span-important. [220]

PRPR 2 (Time-span Segmentation) Let e_k be the prolongationally most important event in a prolongational region (e_i–e_j). If there is a time-span that contains e_i and e_k but not e_j, prefer a prolongational reduction in which e_k is an elaboration of e_i, similarly with the roles of e_i and e_j reversed. [221]

PRPR 3 (Prolongational Connection) In choosing the prolongationally most important event e_k in a prolongational region (e_i–e_j), prefer an e_k that attaches so as to form a maximally stable prolongational connection with one of the endpoints of the region. [224]

PRPR 4 (Prolongational Importance) Let e_k be the prolongationally most important event in a region (e_i–e_j). Prefer a prolongational reduction in which e_k is an elaboration of the prolongationally more important of the endpoints. [226]

PRPR 5 (Parallelism) Prefer a prolongational reduction in which parallel passages receive parallel analyses. [226]

PRPR 6 (Normative Prolongational Structure) A cadenced group preferably contains five elements in its prolongational structure:

a. a prolongational beginning,

b. a prolongational ending consisting of one element of the cadence,

c. a right-branching prolongation as the most important direct elaboration of the prolongational beginning,

d. a right-branching progression as the (next) most important direct elaboration of the prolongational beginning,

e. a left-branching "subdominant" progression as the most important elaboration of the first element of the cadence. [234]

B. XML Snippet for Melodic Phrase in Chopin's "Grande Valse Brillante"

```
<primary>
  <ts timespan="1.0" leftend="5.0" rightend="6.0">
    <head>
      <chord duration="1.0" velocity="90">
        <note id="P1-2-4" />
      </chord>
    </head>
    <at>
      <temp difference="3.0" stable="./../..">
        <pred temp="./../.." />
        <succ temp="+inf" />
      </temp>
    </at>
  </ts>
</primary>
<secondary>
  <ts timespan="2.0" leftend="3.0" rightend="5.0">
    <head>
      <chord duration="1.0" velocity="90">
        <note id="P1-2-1" />
      </chord>
    </head>
    <at>
      <temp difference="2.0" stable="./..">
        <pred temp="-inf" />
        <succ temp="./.." />
      </temp>
    </at>
    <primary>
      <ts timespan="1.0" leftend="3.0" rightend="4.0">
        <head>
          <chord duration="1.0" velocity="90">
            <note id="P1-2-1" />
          </chord>
        </head>
```

```
    <at>
      <temp difference="2.0" stable="./../..">
        <pred temp="-inf" />
        <succ temp="./../.." />
      </temp>
    </at>
  </ts>
</primary>
<secondary>
  <ts timespan="1.0" leftend="4.0" rightend="5.0">
    <head>
      <chord duration="0.5" velocity="90">
        <note id="P1-2-2" />
      </chord>
    </head>
    <at>
      <temp difference="1.0" stable="./..">
        <pred temp="./.." />
        <succ temp="+inf" />
      </temp>
    </at>
    <primary>
      <ts timespan="0.5" leftend="4.0" rightend="4.5">
        <head>
          <chord duration="0.5" velocity="90">
            <note id="P1-2-2" />
          </chord>
        </head>
        <at>
          <temp difference="1.0" stable="./../..">
            <pred temp="./../.." />
            <succ temp="+inf" />
          </temp>
        </at>
      </ts>
    </primary>
    <secondary>
      <ts timespan="0.5" leftend="4.5" rightend="5.0">
        <head>
          <chord duration="0.5" velocity="90">
            <note id="P1-2-3" />
          </chord>
        </head>
        <at>
```

```
            <temp difference="0.5" stable="./..">
              <pred temp="./.." />
              <succ temp="+inf" />
            </temp>
          </at>
        </ts>
      </secondary>
    </ts>
  </secondary>
</ts>
</secondary>
```

C. Generated String for the CFG Before Training

S → N

N → N N

N → m24

N → m23

N → m22

N → m21

N → m20

N → m19

N → m18

N → m17

N → m16

N → m15

N → m14

N → m13

N → m12

N → m11

N → m10

N → m9

N → m8

N → m7

N → m6

N → m5

N → m4

N → m3

N → m2

N → m1

N → 0

N → 1

N → 2

N → 3

N → 4

N → 5

N → 6

N → 7

N → 8

N → 9

N → 10

N → 11

N → 12

N → 13

N → 14

N → 15

N → 16

N → 17

N → 18

N → 19

N → 20

N → 21

N → 22

N → 23

N → 24

0 → m2 2

0 → m1 1

0 → m1 1

0 → 0 0

0 → 1 m1

0 → 2 m2

m4 → m2 m2

m3 → m2 m1

m3 → m1 m2

3 → 1 2

3 → 2 1

4 → 2 2

1 → 3 m2

2 → 3 m1

2 → 4 m2

3 → 4 m1

3 → 5 m2

4 → 5 m1

4 → 6 m2

5 → 6 m1

5 → 7 m2

6 → 7 m1

6 → 8 m2

7 → 8 m1

7 → 9 m2

8 → 9 m1

8 → 10 m2

9 → 10 m1

9 → 11 m2

10 → 11 m1

10 → 12 m2

11 → 12 m1

11 → 13 m2

12 → 13 m1

12 → 14 m2

13 → 14 m1
13 → 15 m2
14 → 15 m1
14 → 16 m2
15 → 16 m1
15 → 17 m2
16 → 17 m1
16 → 18 m2
17 → 18 m1
17 → 19 m2
18 → 19 m1
18 → 20 m2
19 → 20 m1
19 → 21 m2
20 → 21 m1
20 → 22 m2
21 → 22 m1
21 → 23 m2
22 → 23 m1
22 → 24 m2
23 → 24 m1
m23 → m24 1
m22 → m24 2
m22 → m23 1
m21 → m23 2
m21 → m22 1
m20 → m22 2
m20 → m21 1
m19 → m21 2
m19 → m20 1
m18 → m20 2
m18 → m19 1
m17 → m19 2
m17 → m18 1
m16 → m18 2
m16 → m17 1
m15 → m17 2
m15 → m16 1
m14 → m16 2
m14 → m15 1
m13 → m15 2
m13 → m14 1
m12 → m14 2
m12 → m13 1
m11 → m13 2
m11 → m12 1
m10 → m12 2
m10 → m11 1
m9 → m11 2
m9 → m10 1
m8 → m10 2
m8 → m9 1
m7 → m9 2
m7 → m8 1
m6 → m8 2
m6 → m7 1
m5 → m7 2
m5 → m6 1
m4 → m6 2
m4 → m5 1
m3 → m5 2
m3 → m4 1
m2 → m4 2
m2 → m3 1
m1 → m3 2
1 → m2 3
2 → m2 4
3 → m2 5
4 → m2 6
5 → m2 7
6 → m2 8
7 → m2 9
8 → m2 10
9 → m2 11
10 → m2 12
11 → m2 13
12 → m2 14
13 → m2 15
14 → m2 16
15 → m2 17
16 → m2 18
17 → m2 19
18 → m2 20
19 → m2 21
20 → m2 22
21 → m2 23
22 → m2 24
2 → m1 3
3 → m1 4
4 → m1 5
5 → m1 6
6 → m1 7
7 → m1 8
8 → m1 9
9 → m1 10
10 → m1 11
11 → m1 12
12 → m1 13
13 → m1 14
14 → m1 15
15 → m1 16
16 → m1 17
17 → m1 18
18 → m1 19
19 → m1 20
20 → m1 21
21 → m1 22
22 → m1 23
23 → m1 24
m23 → 1 m24

m22 → 1 m23
m21 → 1 m22
m20 → 1 m21
m19 → 1 m20
m18 → 1 m19
m17 → 1 m18
m16 → 1 m17
m15 → 1 m16
m14 → 1 m15
m13 → 1 m14
m12 → 1 m13
m11 → 1 m12
m10 → 1 m11
m9 → 1 m10
m8 → 1 m9
m7 → 1 m8
m6 → 1 m7
m5 → 1 m6
m4 → 1 m5
m3 → 1 m4
m2 → 1 m3
m22 → 2 m24
m21 → 2 m23
m20 → 2 m22
m19 → 2 m21
m18 → 2 m20
m17 → 2 m19
m16 → 2 m18
m15 → 2 m17
m14 → 2 m16
m13 → 2 m15
m12 → 2 m14
m11 → 2 m13
m10 → 2 m12
m9 → 2 m11
m8 → 2 m10
m7 → 2 m9
m6 → 2 m8
m5 → 2 m7
m4 → 2 m6
m3 → 2 m5
m2 → 2 m4
m1 → 2 m3
0 → ‘0’
1 → ‘1’
2 → ‘2’
3 → ‘3’
4 → ‘4’
5 → ‘5’
6 → ‘6’
7 → ‘7’
8 → ‘8’
9 → ‘9’
10 → ‘10’
11 → ‘11’
12 → ‘12’
13 → ‘13’
14 → ‘14’
15 → ‘15’
16 → ‘16’
17 → ‘17’
18 → ‘18’
19 → ‘19’
20 → ‘20’
21 → ‘21’
22 → ‘22’
23 → ‘23’
24 → ‘24’
m24 → ‘-24’
m23 → ‘-23’
m22 → ‘-22’
m21 → ‘-21’
m20 → ‘-20’
m19 → ‘-19’
m18 → ‘-18’
m17 → ‘-17’
m16 → ‘-16’
m15 → ‘-15’
m14 → ‘-14’
m13 → ‘-13’
m12 → ‘-12’
m11 → ‘-11’
m10 → ‘-10’
m9 → ‘-9’
m8 → ‘-8’
m7 → ‘-7’
m6 → ‘-6’
m5 → ‘-5’
m4 → ‘-4’
m3 → ‘-3’
m2 → ‘-2’
m1 → ‘-1’