Generating Music Using Concepts from Schenkerian Analysis and Chord Spaces *

Donya Quick

Department of Computer science
Yale University
New Haven, CT 06520
Email: donya.quick@yale.edu

Yale Research Report 1440

May 10, 2010

Abstract

Music demonstrates structure at many levels. This structure can be found in tonalities as well as in temporal aspects of the music. However, many approaches to algorithmic composition neglect the structure at one of these levels, either requiring existing musical fragments as input or not imposing sufficient global structure. The results of these algorithms have a disorganized or random sound [21, 12]. This paper presents a more balanced approach that considers structure at both levels.

Schenkerian analysis is a method for analyzing classical Western music based on recursive simplification of harmonic structure in compositions at different levels of abstraction [18]. This paper describes a generative grammar for classical Western chord progressions using ideas from Schenkerian analysis. The output from the grammar gives the general structure of a chord progression over time, but is still an abstract representation that requires additional information to become a complete score.

Recent research has attempted to find mathematical relationships between harmonies as a way to model perceived similarity between various chords [3, 20]. This kind of model is necessary to reduce the solution space for individual notes to a manageable number of choices. The structural grammar inspired by Schenkerian analysis is combined with these mathematical models for harmony to improve voice-leadings and create a melodic backbone that requires little modification to produce melodies. Results from an implementation of this approach demonstrate a structured sound and simple melodies while avoiding the need for existing music as input.

*This research was supported in part by NSF grant CCF-0811665.

1 Introduction

Music is highly structured. At the pitch level, structure exists due to conformity to various tonal systems and scales. The tonnetz, an old system of relating similar-sounding triads [14], as well as more recent work into chord geometries [3, 20], has attempted to address this level of structure. Music is also temporally structured. In Western music, compositions are formed in a strongly hierarchical fashion, with long compositions broken into distinct movements and shorter compositions (or movements themselves) broken into numerous sections. At a more local level, melodies can be reduced to collections of smaller phrases or motives. While attention is being increasingly given to the structure that underlies harmonies by examining mathematical relationships between chords [3, 20], attention to temporal structure at a more abstract level in music is often lacking from many algorithms for automating composition.

Towsey and Werner classify generative algorithms for music into three categories: rule-based approaches, systems that learn by example, and systems that incorporate genetic algorithms [10]. Markov chains and fractal-like approaches such as Lindenmayer systems (L-systems) are common subjects for introducing structure [21, 8, 22]. Markov chains represent learnable models, while L-systems are rule-based grammars. Both Markov chains and L-systems are prone to only considering structure at a local level if the states or grammar alphabet consist of concrete notes or simple operations on them. Other algorithms, such as those proposed for contrapuntal music generation by Acevedo, attempt to address structure between voices using genetic algorithms, but require existing musical fragments as input [1].

Creating a reasonable-sounding, original composition that takes input only in the form of random numbers and some combination of music theory and compositional guidelines is a difficult task. Methods such as Schenkerian analysis are appealing from the standpoint of modeling structure at a high level in music. If its main ideas are converted into a generative model rather than an analytical one, this type of approach can solve some of the temporal structure problems occurring in other models. Schenkerian analysis is also attractive due to its use of grammar-like, recursive reductions [18]. Grammars are a common tool in computer science, and their implementation can be broken into two parts: the grammar itself and an algorithm for applying the grammar to generate musical structures. Temporal structure in the music can then be created from the top-down approach.

Determining temporal structure as part of automated composition gives rise to another problem: unless the structure is predetermined down to the level of individual notes, the number of compositions that fit a particular structure will be extremely large to infinite depending on how many musical constraints are imposed. For most human musicians, it is easy to map directly from abstract information about a chord progression, such as "d-G-C," to concrete notes and produce a reasonable compositional result – even if this result differs between individuals. However, this is a difficult task for a machine that lacks a musician's intuition for choosing pitches to form harmonies. On a standard 88-key piano, there are $8 \times 7 \times 7 = 392$ possible ways to play a C-major chord using the pitch classes {C, E, G}. Because of the size of the solution space, even if there is a way to determine good structures for music, selecting and placing notes is still a non-trivial task that requires its own mathematical tools. The chord spaces described by Callender et al. are one such tool that can help reduce the size of the solution space for chord progressions.

1.1 Goals

This paper explores Schenkerian analysis as a candidate model for generating new compositions with the aid of chord spaces to determine voice-leadings, which are the transitions between notes from one chord to the next. Goals were to:

1. Develop generative algorithms from a simplified version of Schenkerian analysis with use of chord spaces.
2. Create Haskell-based implementations for:
   - A grammar-based approach for composing contrapuntal music using Schenkerian analysis as a model.
   - Chord spaces proposed by Callender et al. [3]
3. Test the robustness of the algorithms implemented by generating collections of short compositions.

Schenkerian analysis presents a harmony-centric view of classical Western music that has a large, grammar-like component. A simplified version of this component was used as a way to create a generative grammar for harmonies. Because Schenkerian analysis lacks a formal, algorithmic definition, and because its exact methodology differs between different genres of music, only a single genre was considered for the implementation: simple, contrapuntal classical music that harmonically resembles Baroque fugues, but that lacks the additional melodic constraints imposed by fugue-form. Reasonable documentation of Schenkerian analysis of fugues exists [17], although fugues are a difficult subject from a generative standpoint due to the number of developmental constraints the form imposes. Instead, simpler contrapuntal music is the focus of this paper, specifically short pieces for three voices where each voice carries its own melody. Each of these melodies should, ideally, sound good when heard by itself as well as with the other voices. Because the main difference between a fugue and a more free-form contrapuntal composition is in the melodic development of the composition, documentation of Schenkerian analysis for fugues was considered a sufficient resource for harmonic aspects of more general contrapuntal music.

Generative algorithms were created by attempting to reverse each major step of Schenkerian analysis individually, starting with background generation. The lack of a formal definition of the process for each analysis step meant that some sub-steps and concepts had to be simplified and/or modified to be implemented. As a result, the algorithms and production rules implemented only reflect a simplified subset of the general concepts found in Schenkerian analysis.

Some steps of Schenkerian analysis that are fairly straightforward present significantly higher difficulty from a generative standpoint. For example, it is fairly simple for a human analyzer to label a chord progression in the Roman numeral system, but mapping of a Roman numeral series into a pleasing set of concrete notes is not as easy due to the number of possible mappings involved. Chord spaces were employed as a way to try to solve this problem.

Finally, a Haskell implementation of the algorithms developed was used to test the quality of results and the robustness of the algorithms. The results were generally consonant with reasonable, although very simple melodies within each voice. However, the results did not show much diversity in sound due to the small number of production rules used in the grammar and the simplicity of some of the algorithms.

1.2 Approaches to Harmony Generation

Harmony generation can be broken into two different problems: (1) creating chord progression-style harmonies without melody and (2) creating harmony for an existing melody. The grammar and algorithms in this paper are designed to address the first type of problem. Harmony generation without an existing melody represents a potentially more difficult problem, since a good melody will often already contain elements of harmony in sequential notes, thereby giving a point of reference for any algorithm responsible for adding additional notes.

Many approaches to modeling music for generative purposes only model elements of it rather than a complete process. As a result, the models cannot be used to generate music without significant human intervention. Models that ignore either the temporal or local note-choice aspects of music are both prone this problem. Chord spaces and voice-leading spaces are an example of this kind of musical model [20, 3], since although they map similar chords close to each other, they give little information about how to travel through the space to create music. The information captured in the geometric relationships between the chords is not sufficient to guarantee a good choice for the next chord in a progression. This can be seen to some extent in the results presented by Gogins, which use L-systems to traverse chord space [12]. Compositions from these types of algorithms do not always sound bad, but they do not have a human-like sound. While human imitation may not be the goal for all composition algorithms, results such as these suggest that chord spaces require a more complex supporting algorithm to approach when a human-like sound is desired.

Liangrong and Goldsmith propose a Markov decision process for generating 3-part harmony for exiting melody (yielding 4 voices in total), where states represent transitions between two chords [22]. While this approach may work reasonably given an existing melody for guidance, models without such guidance become prone to two problems: random-sounding progressions due to failure to consider sufficient context (as in the case of the L-systems [12, 21]), or an explosion in the number of states due to trying to consider additional history. For an alphabet of size n and a history of length m, a total of n^m states would be needed. If n must contain every viable combination of pitch classes, this type of representation quickly becomes inefficient. If a more efficient representation could be found, this type of representation would be more useful from the standpoint of learning various properties in music. Clement presents one such approach for generating chord progressions using Markov models [4], but it does not consider any sort of global structure for the progressions - a feature that would be needed to generate larger, complete compositions.

Simple grammars are a common approach to music generation, although not always with good results. Both context-free and context-sensitive algorithms have been explored, but it is difficult to incorporate adequate music theory into relatively simple grammars if they are concerned with note-level generation. Common problems associated with music generated by these simple, grammar-based algorithms are a lack of detectable meter, poor harmonies, and incoherent melodies [21]. Some of these problems may be alleviated by building a grammar based on an existing analysis method that does not try to address note-level detail.

1.3 Schenkerian Analysis

Schenkerian analysis is a method for determining the overall harmonic structure of a composition. Schenker theorized that, at least for large categories of Western classical music, everything was essentially reducible to a I-V-I transition, called the ursatz [17]. The analysis addresses three levels of abstraction in music: the foreground, midground, and background. The foreground is the complete score, and the background is a condensed summary that follows the ursatz. The midground is an intermediate level between the background and foreground. Complex harmonic phrases are considered reducible to simpler ones that serve the same structural purpose. The first step of the analysis distills harmonic elements out of more complex melodic interactions [16]. After this, harmonies are recursively reduced until an overall pattern of I-V-I becomes clear.

Since melodic elements are recursively reduced to harmonies, this method of analysis is distinctly harmony-centric and may not reflect the compositional process in every type of music. For example, Eastern music that utilizes a drone instrument to maintain a constant key throughout a song might require a different method of analysis and generation. Still, although there are some types of music where Schenkerian analysis may be an ill-suited method, for a number of classical Western styles, harmony is extremely important and, therefore, the approach is a good fit.

One major problem with Schenkerian analysis from a generative standpoint is that it is not a concretely defined process. There is little easily-accessible information regarding the detail of the midground reduction rules, and rules are commonly shown by example rather than stating them in precise terms. As a result, despite attempts to formalize portions of the analysis algorithmically [16], at some levels Schenkerian analysis is still not formalized. In order to define an analysis method that succeeds on every piece of music for a given classical style or even composer, it is often necessary to form the rules around the particular genre or style under consideration.

Kemal Ebcioglu developed one method for incorporating a Schenkerian approach into harmony synthesis, but Schenkerian ideas were only employed as a method to check generated harmonies for correctness rather than being used directly for generation [7]. This paper also did not consider use of Schenkerian analysis alone for harmony generation, citing that doing so would easily become stuck and require backtracking to produce a progression that is sound by Schenkerian rules. Because this paper is only concerned with generating chord progressions and does not need to consider existing melodic constraints, these backtracking and efficiency problems are avoided. If the set of generative rules is carefully constructed (which would be necessary if the results are to meet the requirements of a particular genre), there should be little or no need for backtracking to produce reasonable harmonies directly from a Schenkerian-like system of reduction rules.

2 Reversing Schenkerian Analysis

Schenkerian analysis can be roughly broken down into the following steps:

1. Foreground analysis:
   (a) Identify purely melodic elements, such as passing tones and neighboring tones.

   (b) Rhythmically and melodically simplify the music to further extract chordal elements.
   (c) Identify and label chords in the Roman numeral system.
2. Midground analysis: recursively reduce harmonic passages to their representative chords (e.g. IV-V may reduce to V).
3. Background analysis: once a sufficiently small number of chords exist, identify the ursatz.

This process lends itself to reversal of each step into a generative process. Because the boundaries between the steps are fuzzy and dependent on the type of music being considered, there is more than one way to turn these analytical steps into generative steps. The following is a simple approach that isolates each step as its own algorithm.

1. Background generation: start with the ursatz (or something like it for more versatility). This defines the very basic harmonic structure of the composition. For example, if a composition (or passage) follows an A-B-A format, part A may be in the tonic (I) while part B may be in the dominant (V), thereby giving an overall I-V-I structure.
2. Midground generation: recursively expand the ursatz using production versions of reduction rules. This defines the harmonic progression within each larger section of the composition. Each section resulting from background generation will become its own chord progression. Midground generation fills in this finer structure.
3. Foreground generation: introduce melodic elements like passing and neighboring tones into the progression to create implied melodies.

When considering each step described above as an algorithm, input and output at each stage should be the following (further illustrated by figure 1):

1. Background generation
   - Input: information about the constraints of the intended form of music.
   - Output: an outline of the composition's most general level of harmonic structure.
2. Midground generation:
   - Input: the most general level of harmonic structure for a composition.
   - Output: a large set of chords represented as concrete notes instead of chord types.
3. Foreground generation:
   - Input: a progression of concrete notes with limited rhythmic content.
   - Output: a complete score of music with melodic elements (although this is not required to include performance details like dynamics and tempo changes).

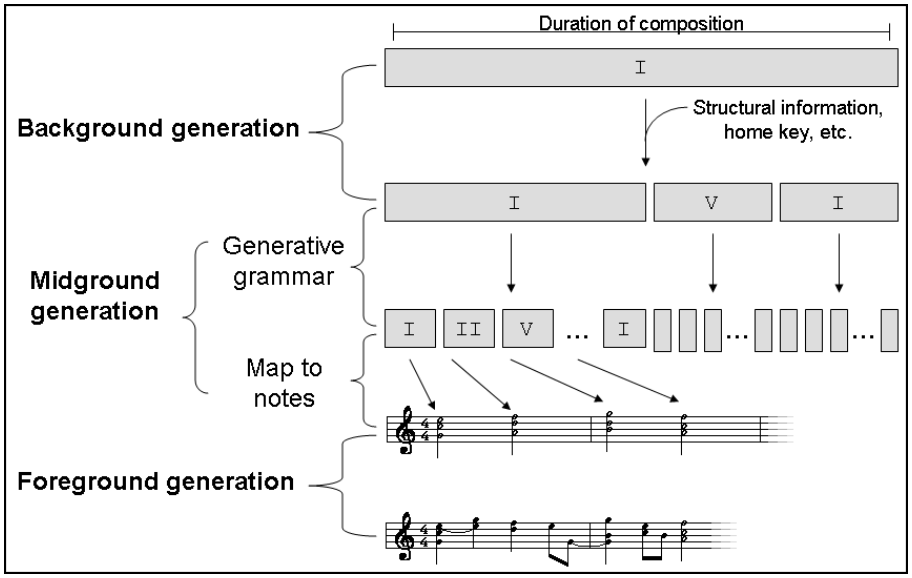

Figure 1: Illustration of the overall process of background generation, midground generation, and finally foreground generation.

The process of background analysis is determined largely by the constraints of the form of music. Midground generation lends itself to representation as a nondeterministic grammar. While the process of background analysis and even midground analysis are relatively well-defined for certain types of music, the process of foreground analysis (and even when foreground analysis stops and midground analysis begins) are not as well defined from an algorithmic standpoint. As a result, the earlier steps of generation are more straightforward than later steps.

2.1 Excluded Aspects of Schenkerian analysis

A complete Schenkerian analysis of a composition contains more information than just the reduction of the harmony to a small string of Roman numerals. Additional notations exist to outline the flow of important harmonic and melodic elements, as well as to locate key harmonic structures in more detail than the ursatz alone. One such structure that Schenkerian analysis tries to locate is an overall structure of a descending scale indices in a given passage of music [17]. This is captured in a concise summary of each section of the music, referred to as a voice-leading matrix [17]. This aspect of the process was ignored here, since the method for locating these descending features was not concretely enough defined to be easily reversed. Use of chord spaces may present a way to re-introduce a voice-leading matrix, but that possibility was not explored here.

Another omitted concept from Schenkerian analysis is the distinction between melody and accompaniment. Melodic lines are thought to follow more linear patterns than supporting voices [17]. Because the type of music considered for the approach and implementation consists of three simultaneous melodies rather than a melody and harmonic accompaniment, these sorts of distinctions between voices were not made.

3 Approach to Background Generation

In classical music, the background of a Schenkerian analysis is essentially the ursatz with some embellishment. A complete analysis retains more information than simply the overall harmonic structure, but only harmonic structure is considered here. Handling of concrete notes is left to later steps of the generative process.

The highest level of harmonic structure can be somewhat determined by the structure of the music. The degree to which the harmony is constrained depends on the type of music. In a fugue, the rules for developmental form can impose harmonic restrictions such as the following although exact interpretations of the structural requirements vary between composers [13]:

- Exposition:
  - Short tonic section
  - Short dominant section
  - Short tonic section
- Middle: variable-length section in a closely-related key, such as the relative major or minor.
- Final section:
  - Short sections in related keys (such as the subdominant or dominant).
  - Closing tonic section.

For some classical forms, as illustrated by the fugue form above, much of the background at the Roman numeral level comes from the expected structure of the music. For simple contrapuntal music with forms similar to the one above, there is not much generation to do at this step. Other than the developmental form, the only other main choices at this point would be less musical and more technical: a key for the tonic, the number of voices involved, any restrictions on the voices' ranges that would need to be considered during subsequent generative steps, etc.

The main musical responsibility of background generation is choice of overall structure and the key of each large section. For short, simple music in a variety of styles (from simple classical to modern popular music), some variation of A-B-A form is common, such as A-A'-B-A. The A sections would be in the tonic, and B would usually be in either the dominant, subdominant, or relative major/minor. Variations of sections (denoted with a ') are usually in the same key as the originals. The primary difference expected between sections A and A' would be in the foreground: slight melodic variations in the A' section that still show some resemblance to the original A section. This suggests that the background for part A' should look at a minimum very similar, if not identical to that of part A. However, the labeling of A and A' as distinct sections becomes important for later steps, since the foregrounds of A and A' should be different.

Since background generation as presented here is a simplified reversal of background analysis, it is largely a matter of choice of form and instrumentation and, therefore, it is the easiest step to adapt to different genres of music. In this case, the same background could be used to create a classical Western composition or a contemporary Jazz piece. Because of this, no specific algorithm was developed for background generation, and it is assumed that the style of music (and therefore the basic developmental pattern) and number of voices can be supplied as human

input or chosen at random from a large table. The approach chosen for the implementation was to leave this step to human input, and will be discussed further in the implementation section.

As will be addressed in the section on Midground generation, it may be possible to make structural decisions part of the grammar responsible for midground generation, thereby combining the two steps into one if a lookup table for desirable backgrounds exists. However, this possibility has not been explored much yet.

4 Approach to Midground Generation

Midground analysis follows a grammar-like process of reducing harmonic structures [18]. It therefore makes sense for midground generation to use a grammar to produce the overall harmonic structure of a composition. Since both background and midground analysis aim to represent a composition by using of a string of Roman numeral chord types (I, II, etc.) with some additional notations, a grammar to represent this process should use similar symbols. Nondeterministic grammars are a good choice for this step. A single non-terminal can have many rules associated with it, and the algorithm applying the grammar is responsible for choosing which one to apply at any given time.

The boundary between foreground and midground for the purposes of generation was considered to be a melodic backbone: a chord progression that contains enough rhythmic and harmonic detail that foreground generation will have little work to do. This means that midground generation is be responsible for some of the fundamental structures needed for reasonable melodies. Because of this, the process described here mill need to be be at least genre-specific, if not also composer-specific. Although the overall process midground generation may be similar for different types of music, the production rules for the grammar-based generative step would likely differ from one genre to another. The grammar implemented was intended to be consistent with early classical Western music, which is more harmonically restrictive than modern music. This does not preclude the grammar from being used to create other styles of music, but rather means that other styles of music may require larger and more complex grammars to have the desired style of harmony. The grammar developed here is also context-free, while a greater diversity of harmony may require a context-sensitive grammar instead.

A generative grammar for Roman numerals alone will not yield complete midground generation, since at some point Roman numerals must be converted into concrete notes. Foreground generation cannot proceed directly from a string of numerals, so midground generation must also be responsible for determining the exact notes for each chord. There are multiple possible ways to approach this problem. The approach taken in this paper keeps the problem of mapping to concrete notes separate from that of the generative grammar. This simplifies the responsibilities of the grammar-based step, but requires an additional algorithm for mapping Roman numerals to concrete notes. Because of this two-step process, midground generation is not an exact reversal of midground analysis (which considers concrete notes in addition to Roman numerals at each step) and the grammar portion of the generation is more simplified.

4.1 Midground Grammar Elements

This section describes the notation, terminals, non-terminals, and the form of the production rules for the grammar-based portion of midground generation. Because midground generation becomes genre specific to some degree, this approach will only be suitable to the chosen genre for implementation and related styles of classical music. Genres like jazz would require modified grammars with more terminals for additional types of chords unless a lot of harmony-related responsibilities were left to the foreground generation algorithm. Similarly, more modern classical music would also require an expanded set of terminals.

4.1.1 Notation

The following notation will be used to describe the grammar and its use.

- Any string of symbols: ϕ.
- Production rule: $S_0 \rightarrow S_1 S_2 ... S_n$. This means that symbol S_0 can be replaced by the string $S_1 S_2 ... S_n$ by an algorithm applying the generative grammar.
- Evaluation: $S_0 \Rightarrow S_1 S_2 ... S_n$. S_0 is evaluated and replaced by the string $S_1 S_2 ... S_n$ when some production rule is applied.
- Evaluation of a string of symbols by a particular algorithm: $A(\phi) \Rightarrow \phi'$. Given algorithm A with parameters p for applying the grammar: $A(\phi, p) \Rightarrow \phi'$ refers to the evaluation of ϕ by A supplied with p.
- Chord types: Roman numerals I through VII represent the seven possible chord types given a home key and mode. Major and minor chord types are not distinguished by this simple grammar, since the source material used to derive production rules, [17], did not distinguish them in its examples. This is a limiting factor on the degree of harmonic detail that can be recorded in production rules, although the results showed that it was mostly sufficient for simple music with Baroque-era harmonies.

4.2 Terminals and Non-Terminals

To create a grammar for the Roman numeral system of labeling chord types, terminals and non-terminals must be defined to handle the following musical elements: the "types" of chords (captured by their Roman numerals), the durations of chords, and modulated series of chords.

4.2.1 Chord Type Non-Terminals

The non-terminals are I, II, III, IV, V, VI, and VII. Each of these represents the relative one, two, etc. chord of a given scale. For this version, any non-terminal may be either major or minor depending on the scale. Miscellaneous chord type non-terminals will be denoted by C. To keep track of the duration of a non-terminal, subscripts of the form C_t are used, where t is the duration of chord C.

Schenkerian analysis assumes that a series of concrete chords can serve the musical purpose of another, single chord of the same duration. Chord type non-terminals are placeholders representing such sections. A non-terminal of I_t represents t amount of time to be filled by material for which a I chord is representative.

4.2.2 Chord Type Terminals

The terminals are Ic, IIc, IIIc, IVc, Vc, VIc, and VIIc. The "c" appended to each of the chord types simply indicates that it represents an actual chord (terminal) rather than a possible grouping of other chords (non-terminal). Miscellaneous chord type terminals will be denoted by c, and durations will be represented in the same way as for non-terminals. The notation c_t indicates that chord type terminal c has a duration of t.

Unlike the non-terminals, chord type terminals are placeholders for sets of possible concrete notes. The Roman numeral system of labeling harmony in music is based on classifying chords as triads. If a scale is represented as a series of indices, then the triad-based definition of each Roman numeral is the following:

Numeral	Scale indices	Mode (major tonic)	Mode (minor tonic)
I	0,2,4	Major	Minor
II	1,3,5	Minor	Minor/diminished
III	2,4,6	Minor	Major
IV	0,3,5	Major	Minor
V	1,4,6	Major	Minor
VI	0,2,5	Minor	Major
VII	1,3,6	Minor/diminished	Major

Table 1: Triad definitions of Roman numerals. The scale indices for a given chord may occur in any octave.

4.2.3 Modulation Terminals

In Schenkerian analysis, modulated sections are considered to be representative of other chords in the tonic. In analysis, this implies that a section that modulates to the subdominant may be representative of IV in the tonic. From a generative standpoint, this suggests that replacing some duration of IV with another progression in the subdominant is a reasonable tactic to achieve greater harmonic diversity.

When harmony rules for music are strict, only notes within the current key are considered valid choices for harmony, although this limitation is not necessarily imposed on ornamental foreground elements, such as grace notes. For example, in early classical music, F# would not be an acceptable note to harmonize with a melody in the key of C-major, because F# does not occur within the scale and thus would sound dissonant. However, if one modulates from C-major to G-major (modulation to the dominant when C-major is the tonic), F# becomes part of the scale. This is illustrated by table 2.

Short modulations can be employed to add brief harmonic diversity without completely changing the tone of a passage of music and the overall sense of what the tonic is. When one modulates to the "relative (some number)" of a key (or the subdominant, dominant, etc.), it implies generating a new scale based on the following rules, if major and minor are the only modes considered. This is illustrated

Relative chord name	Triad notes	Scale type
Tonic	C, E, G	Major
Relative 2^{nd} (supertonic)	D, F, A	Minor
Relative 3^{rd} (mediant)	E, G, B	Minor
Relative 4^{th} (subdominant)	C, F, A	Major
Relative 5^{th} (dominant)	D, G, B	Major
Relative 6^{th} (relative minor or sub-mediant)	C, E, A	Minor
Relative 7^{th} (subtonic)	D, F, B	Minor

Table 2: Examples of the implications of modulations to keys from C-major when the only available modes are major and minor.

by table 3.

Modulation #	Terminal	New mode (from maj.)	New mode (from min.)
Relative 2^{nd}	Mod-II(...)	Minor	Minor
Relative 3^{rd}	Mod-III(...)	Minor	Major (rel. major)
Relative 4^{th}	Mod-IV(...)	Major	Minor
Relative 5^{th}	Mod-V(...)	Major	Minor
Relative 6^{th}	Mod-VI(...)	Minor (rel. minor)	Major
Relative 7^{th}	Mod-VII(...)	Minor	Major

Table 3: The mapping of modulation types to scale types considering only major and minor modes. When the tonic is minor, Mod-III will result in a major key (the relative major), and when the tonic is major, Mod-VI will result in a minor key (the relative minor). These two modulations are unique since they do not change the key signature, only the perception of the home key.

With the exception of modulations to relative major/minor keys, each modulation implies a change in key signature with this definition of modulations. This is assuming that other modes like Dorian are not allowable, which is somewhat limiting even for Baroque music. Fortunately, for the type of music considered here, modulations to keys other than the subdominant, dominant, and relative major/minor would be rare (although they were allowed with low probability in the implementation). For the subdominant and dominant, modulation to major or minor keys is common.

To avoid problems that can occur with successive key changes, modulations occur as only terminals (unlike Roman numeral chord types, for which each non-terminal has a corresponding terminal). Six types of modulations will be considered, one for each scale index other than the root: Mod-II, Mod-III, Mod-IV, Mod-V, Mod-VI, and Mod-VII. Mod-II indicates a key change to the relative second, Mod-III to the relative third, etc. The notation Mod-c will represent an arbitrary modulation.

When a modulation is employed for a section, it is important to distinguish which chord terminals and non-terminals fall under the modulation and which do not. Parentheses are additional terminals used to serve this purpose. The notation Mod-c(ϕ) indicates that the string ϕ is to be interpreted in context of the new scale

implied by Mod-c. Once the parentheses are left, the key returns to what it was prior to the Mod-c terminal.

Nested modulations are potentially problematic in this system without any additional constraints imposed, and, therefore, either production rules must be carefully constructed where modulations are concerned, or the algorithm applying the grammar must give special consideration to modulations. If the production rules are poorly constructed or the algorithm applying the grammar does not handle modulations carefully, it might be possible for the following to occur:

Ic (Mod-V (Mod-V (Mod-V (Mod-V (Mod-V (Mod-V (Mod-V(Ic))))))))

In this example the circle of fifths is followed by the modulations. In the key of C-major, this would result in a C-major chord followed immediately by a C#-major chord, which would be considered a bad transition in most traditional classical music. There are two possible fixes for this problem:

1. Make modulations very unlikely to be used by the algorithm applying the grammar.
2. Enforce that some terminals occur at the start and end of a modulation when it is chosen. Instead of allowing a production of the form $C_t \rightarrow$ Mod-$c(I_t)$, only allow productions of the form $C_t \rightarrow$ Mod-$c(c_{t1}...C_t...c_{tn})$, such that the transition in and out of a modulation is more controlled by terminals.

By employing both of these tactics when constructing production rules, it becomes less likely that undesirable key changes will occur.

4.2.4 Production Rules and Algorithm for Grammar Application

Unless the list of production rules is excessively large, some degree of genre-specificity will be involved in their construction. Assuming that well-defined reduction rules exist for the target genre, conversion of reduction rule examples to production rules is fairly straightforward and only considers the Roman numerals. For an example where I-II-V-I is considered representative of I, the corresponding production rule would be of the form:

$\mathrm{I}_t \rightarrow \mathrm{I}_{t1}\ \mathrm{II}_{t2}\ \mathrm{V}_{t3}\ \mathrm{I}_{t4}$

where $t = t1 + t2 + t3 + t4$. Once a set of rules of this form exist, an algorithm is required for applying the rules. Some kind of probabilistic approach would be required for variation. It is also possible that backtracking might be useful for this kind of algorithm, although it was not considered here. The implemented set of production rules and grammar-application algorithm required many genre-specific choices that are detailed in later sections.

4.3 Chords to Concrete Notes

Regardless of the exact algorithmic choices for grammar application, it is reasonable to assume that the choice of exact notes to represent a chord should be somewhat independent of the choice of the type of chords used in production rules. Without this separation, it would be difficult to define a set of generalized production rules

for chord progressions due to the number of possible productions. The following are two such possible ways to include the step of mapping chord type terminals to notes into mid-ground generation:

1. Generate the entire chord progression first as a string of chord type terminals, then map each terminal to a set of concrete notes.
2. Interleave choice of concrete notes with application of production rules. An algorithm taking this approach could potentially use choices of concrete notes when considering which production rule to apply next, but the production rules would not need to consider concrete notes.

The first approach was the one chosen for the implementation. This decision was made because it is the simpler of the two approaches. However, the second approach is closer to an exact reversal of midground analysis, since it re-introduces the notion of a voice-leading matrix. This alternative approach may be worth pursuing in future implementations.

Regardless of whether the first or second approach is taken to choosing concrete notes for each terminal, there are several considerations for this type of algorithm that make it a non-trivial task. Since the harmony is seen as a melodic backbone in Schenkerian analysis, the result of midground generation must have certain properties that make it a good candidate for addition of melodic elements. Some desirable traits would be:

- Variation in chord stackings. For example, it would be very boring for the same voice to always have to play the root of each chord.
- Reasonable spacing of the voices, such that they are neither "too close" nor "too far apart."
- Voices that do not always move in the same direction.

Exactly how much of this should be the responsibility of midground generation versus foreground generation is unclear and presumably would vary by genre. However, because of the notion of midground containing a melodic backbone, it was assumed for the implementation that the midground should produce results that sound "almost" melodic, but not quite, in voices intended to carry a melody (which is all voices in the case of contrapuntal music). This makes foreground generation a simpler task.

4.4 Use of Chord Spaces

Callender et al. introduce five relations on points in pitch space that are based on concepts in music theory. [3]

1. Octave equivalence, O. Chords with n voices belong to the same equivalence class if they have the same order of pitch classes. The fundamental domain lies in the range $[0, 12]$ along all n axes in pitch space. Mathematically this is a relation defined on vectors: $v \sim_O v + (12i),\ i \in \mathbb{Z}^n$.
2. Permutation equivalence, P. Chords with n voices and same pitch content, regardless of order, belong to the same equivalence class. Equivalence class labels for P are multiset, since duplicate elements may exist. P can be defined using the symmetric group of order n (S_n): $v \sim_P \sigma(v),\ \sigma \in S_n$.

3. Transposition equivalence, T. Chords with n voices and the same intervallic content belong to the same equivalence class. The relation is defined on vectors as $v \sim_T v + c,\ c \in \mathbb{R}$. This may be restricted to $c \in Z$ if pitch space is considered to be discrete, as is the case for the implementations discussed in this paper.
4. Inversion equivalence, I. Chords are related to their structural inverses. This can be used to relate major and minor chords when I is used in conjunction with O, P, and T.
5. Cardinality equivalence, C. Chords with different numbers of voices are related.

Unless constrained in some way (such as restricting range based on voice), pitch space for n voices includes every possible ordered combination of n notes. Each n-voice chord can be plotted as a point in n dimensions, with one voice per axis.

To be useful, all of the $OPTIC$ operations must be equivalence relations [3]. However, the definitions for I and C given by Callender et al. are problematic, since I is defined as $x \sim_I -x$ and C is defined as $\{..., x_i, x_{i+1}, ...\} \sim_C \{..., x_i, x_i, x_{i+1}, ...\}$ [3]. Although inversional and cardinality equivalence are useful in concept, the definitions given by Callender et al. are not reflexive and, therefore, are not equivalence relations. Because of this discrepancy, inversion and cardinality functions are not used here.

The most useful operations for this paper's goals were O, P, and T, all of which are equivalence relations. Proofs of various properties for these relations can be found in appendix A. Each of these operations can be considered either as applying to individual chords regardless of context, or they can be applied uniformly to entire progressions. Using equivalence classes to simplify harmonic relationships can yield better ways to produce progressions with reasonable spacing between the voices and reasonable voice-leadings.

4.4.1 Testing Equivalence of Two Chords

Equivalence of two chords, $x = [x_1, ..., x_n]$ and $y = [y_1, ..., y_n]$ for the OPT relations individually can be found by the following checks:

- $x \sim_O y$ if $[x_1 \ mod\ 12, ..., x_n \ mod\ 12] = [y_1 \ mod\ 12, ..., y_n \ mod\ 12]$
- $x \sim_P y$ if $\{x_1, ..., x_n\} = \{y_1, ..., y_n\}$ when considering x and y as multisets.
- $x \sim_T y$ if $[x_1 - x_1, x_2 - x_1, ..., x_n - x_1] = [y_1 - y_1, y_2 - y_1, ..., y_n - y_1]$

For O, P, and T used individually, testing equivalence of two points in pitch space is simple. However, checking equality between two points under multiple equivalence relations can be non-trivial. Two points, x and y, are equivalent under some set of equivalence relations, $R_1, R_2, ..., R_n$ if the following is true:

$$\exists z_1, z_2, ..., z_{n-1}.\ \ x \sim_{R_1} z_1, z_1 \sim_{R_2} z_2, ..., z_{n-1} \sim_{R_n} y$$

Finding the intermediate points, $z_1, ..., z_{n-1}$, required to relate x and y may be difficult unless entire equivalence classes are searched at each step. It would be more efficient to test equivalence between two points with a function that does not require searching. While brute-force searching solutions will guarantee an eventual

correct answer, they will only work on discrete spaces. Although only the discrete version of pitch space is used in the implementations discussed in this paper, it may be desirable to consider continuous spaces in future work. This necessitates mathematical shortcuts for checking equivalence between two vectors over multiple relations.

O-equivalence of two points in pitch space has an easy mathematical shortcut: take each vector mod 12 and check equality. Similarly, P-equivalence is straightforward. Since only the multiset content of two points in pitch space is important, vector equality can be used once each set is sorted in ascending order and checked for equality. However, even if commutativity holds for relations R_1 and R_2 such that $x \sim_{R_1 R_2} y$ implies $x \sim_{R_2 R_1} y$, the existential intermediates needed to relate x and y may be different. This also means that algorithmic shortcuts for checking $x \sim_{R_1} y$ and $x \sim_{R_2} y$ may not be commutative for checking $x \sim_{R_1 R_2} y$ or may not even be suitable in any order. For example, consider the following case of checking $[7, 4, 0]$ and $[60, 4, 7]$ for OP-equivalence with these individual tests of equality using Haskell functions:

$(sort.(map\ (`mod`12)))\ [60, 4, 7] \mapsto [0, 4, 7]$
$(sort.(map\ (`mod`12)))\ [7, 4, 0] \mapsto [0, 4, 7]$

This is a success, but if we reverse the order of operations:

$((map\ (`mod`12)).sort)\ [60, 4, 7] \mapsto [4, 7, 0]$
$((map\ (`mod`12)).sort)\ [7, 4, 0] \mapsto [0, 4, 7]$

Performing the operations in this order will not work to test for OP-equivalence, since O operations have the ability to change which element in a vector is the largest. Fortunately, the shortcut for testing OP-equivalence without searching entire equivalence classes is still simple: always apply the O relation by taking the vector mod 12 before applying the P relation to sort the vector.

OPT equivalence demonstrates a similar problem, but it cannot be solved by simply fixing the order in which operations are performed. Chords are most easily compared for OPT-equivalence when they occupy the smallest range or are in their most compact form, which can be accomplished by using octave shifts. For some chords, simply placing each voice in the range a single octave, sorting the vector, and shifting it to a normalized position will be sufficient. However, other chords may have more than one stacking of equal range when using O operations. For example, $[0, 4, 8]$ and $[12, 4, 8]$ fill the same range, as do $[0, 5, 7]$ and $[12, 5, 7]$. As a result, chords either need to be both placed into standardized stacking forms or all stackings of one chord must be checked against the other. The following approach can be used to test for OPT-equivalence without searching through all members of an equivalence class:

Algorithm 1 *Checking OPT-Equivalence.*

1. *For input vectors v_1 and v_2 of length n, find v_1' and v_2' from v_1 and v_2 respectively by:*
   (a) *One application of O: z_1 = the input vector mod 12.*
   (b) *One application of P: $z_2 = sort\ z_1$.*

(c) *One application of T: $v_1' =$ shift z_2 down by a constant such that the first voice is 0.*

2. *Find S, the set of all sorted, T-normalized stackings of v_2'. Calculate element $s_i \in S$ (where $i \in [0, n]$) by the following:*
   (a) *Apply O to v_2' with the octave shift vector represented by the string $1^i 0^{n-i}$.*
   (b) *Sort and then normalize the resulting vector so that the first voice is 0. This uses one application of P and T respectively.*
3. *Check v_1' for equivalence against all elements in S'. If a match is found, v_1 and v_2 are OPT-equivalent, otherwise they are not.*

For example, if $v_1 = [60, 2, 7]$ and $v_2 = [5, 0, 67]$, $v_1' = [0, 2, 7]$, $v_2' = [0, 5, 7]$, and $S = \{[0, 5, 7], [0, 2, 7], [0, 5, 10]\}$. Because $v_1' \in S$, v_1 and v_2 can be considered OPT-equivalent.

While it is possible to relate two chords with only one application of O, P, and T, it can be complicated to find the correct intermediate steps. Appendix A shows that multiple applications of relations in OPT can be compressed to a single application of each type of relation. Therefore, given a series of operations in OPT connected with function composition, it is possible to find a single operation of each type (one from O, one from P, and one from T) that achieves the same result.

4.4.2 OPT-Equivalence and the Roman Numeral System

When the O and P equivalence relations are combined and applied to individual chords, the resulting equivalence classes are represented by multisets of pitch classes. When applied to individual chords, all C-major triads belong to class $[0, 4, 7]$, all D-major triads to class $[2, 6, 9]$, etc. There will also be a lot of other classes such as $[0, 1, 2]$, which represents the pitch classes $\{C, C\#, D\}$. While OP-equivalence clearly represents one way in which musicians think about harmonies, it does not directly correlate to the Roman numeral system. Ignoring the fact that there are too many equivalence classes, there is no notion of each chord's home key. Without some additional level of classification, we don't know whether the OP class of $[0, 4, 7]$ represents a I-chord in C-major, a V-chord in F-major, etc.

T can also be applied to individual chords with OP-equivalence. The result is that all major chords, regardless of exact pitch class content, will fall into the same equivalence class. Although this is yet another way that musicians consider harmonies, it is actually too abstract for the purposes of the Roman numeral system. With application of OPT to individual chords, all major numerals will be classified under the class represented by $[0, 4, 7]$.

Fortunately, each of these operations has the possibility of being applied at the individual chord level or uniformly to a progression of chords. If OP-equivalence is applied to individual chords and T is applied uniformly at the progression level, the right level of abstraction can be achieved. Since Roman numerals refer to chords relative to some reference note (or home key), a progression in Roman numerals simply defines the progression as a set of possible paths through pitch space. Exactly which path is selected depends on the key and starting point. This level of abstraction is accomplished by assuming use of progression-level T until the end of midground generation (or even the end of foreground generation), at which point the progression can be transposed into the desired key using a mode and reference note.

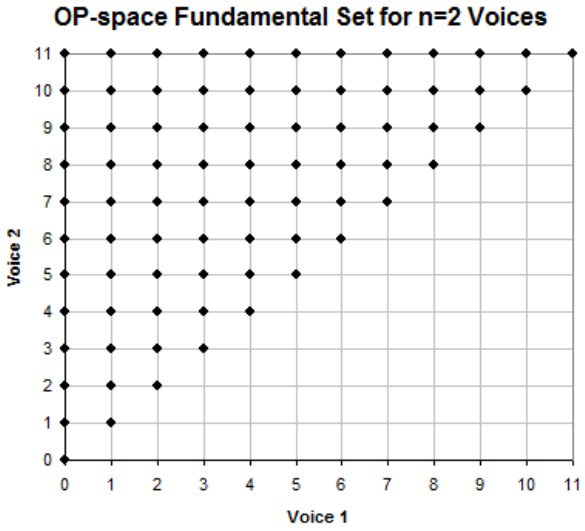

Figure 2: The fundamental domain for OP-equivalence in discrete pitch space for $n = 2$ voices. The equivalent space for $n = 3$ voices forms a tetrahedron with the space for $n = 2$ as the base.

Using pitch numbers, the relationship between Roman numerals and OP-space for individual chords with T-equivalence at the progression level is the following:

1. $[0, 3, 7]$ = minor I (minor scale), usually denoted as i
2. $[0, 4, 7]$ = major I (major scale)
3. $[2, 5, 8]$ = diminished II (minor scale), usually denoted as ii
4. $[2, 5, 9]$ = minor II (major scale), usually denoted as ii
5. $[3, 7, 10]$ = major III (minor scale)
6. $[4, 7, 11]$ = minor III (major scale), usually denoted as iii
7. $[0, 5, 8]$ = minor IV (minor scale), usually denoted as iv
8. $[0, 5, 9]$ = major IV (major scale)
9. $[2, 7, 10]$ = minor V (minor scale), usually denoted as v
10. $[2, 7, 11]$ = major V (major scale)
11. $[0, 3, 8]$ = major VI (minor scale)
12. $[0, 4, 9]$ = minor VI (major scale), usually denoted as vi
13. $[2, 5, 10]$ = diminished VII (major scale), usually denoted as vii
14. $[2, 5, 11]$ = major VII (major chord, minor scale)

4.4.3 Modulations using the OPT relations

The OPT relations can also help to simplify the way in which modulations are handled. A string generated by the Roman numeral grammar that has modulations can be represented as a list of points in the fundamental domain of OP-space with assumed T at the progression level. However, modulation information must be retained if Ic is to be interpreted differently from Mod-V(Ic). One possible

solution is to switch from considering a reference note for the entire progression to considering a reference note for each individual chord. If the pitch class content for a chord is dictated by a pair containing the relevant point in OP-space and a reference note in halfsteps for use with T, we could rewrite the progression Ic in the tonic as $([0, 4, 7], 0)$ and Mod-V(Ic) from the tonic as $([0, 4, 7], 7)$. When transposed, $([0, 4, 7], 7)$ would be mapped to $[2, 7, 11]$ in OP-space. This method of handling the OPT equivalence relations was used for handling modulations in the implementation discussed in later sections, since it allows for representation of an entire progression in OP-space without the nested structures resulting from modulation terminals.

5 Approach to Foreground Generation

For genres where the idea of midground as a melodic backbone is reasonable, it would make sense that foreground generation should not completely obscure the basic voice contours and chord progressions established by midground generation. Compared to the step of foreground analysis, the melodic portion of the foreground generation algorithm in the implementation is quite simple. As a result, it will not construct anything but simple melodies.

Although not an exhaustive list, the following are some general melodic elements that a foreground generation algorithm should be responsible for:

- Passing tones. These are notes leading from one chordal tone to another.
- Neighboring tones. These notes form local minima/maxima above or below a chordal tone.
- Anticipations. A note is "anticipated" when it is played prior to being settled on.
- Suspensions. A note is carried across more than one chord without repeats.
- Note substitutions. Another note may be able to replace a chordal tone while still retaining the same general sound.
- Note removal/adding rests. It is not always necessary for all voices to be playing at the same time. Rests can add to rhythm and help to shape melodic phrases.
- Ornamentation. Some examples of these are trills and grace notes.

Handling of each of these elements would vary by genre and many would require consideration of musical context. For example, harmony choices in other voices must be considered when modifying notes to avoid undesirable sets of notes, such as clusters of major and minor seconds. Rhythm would also become important unless it is already largely defined by the results of midground generation. In the implementation for simple contrapuntal music, rhythm was not considered except at a very basic level.

6 Haskell Implementation

This section describes implementations of the general methods already described for reversing Schenkerian Analysis. This implementation was done in Haskell and is limited to the target genre of simple contrapuntal music for three voices.

6.1 Background Generation Implementation

Since background generation is largely determined by the genre and may have some arbitrary decisions, the background was predefined for this implementation. The format of A-A'-B-A was used with the following stipulations for each section:

Section	Midground	Foreground
A	8 measures of tonic progression	foreground 1
A'	8 measures, same as A	foreground 2 (different from A)
B	8 measures of dominant progression	foreground 3

Three values are taken as user input: a random number seed, a tonic mode, and a reference note for the tonic. The reference note for the dominant scale is assumed to be seven half steps higher than the reference note for the tonic, although this decision was somewhat arbitrary. The expected range for B sections is therefore slightly higher than for A and A' sections.

The A and B midgrounds are generated separately, and the A, A', and B foregrounds are generated separately. No special consideration is given to transitions between sections. The number of voices, although fixed to be three, is determined by midground generation.

6.2 MG1: Midground Generation Implementation without Chord Spaces

The method for midground generation described in this section will be referred to as MG1. Midground generation in MG1 requires the following input: a starting Roman numeral string, a length, and a string of random numbers. The Haskell implementation has the following interface for generating a midground:

```
> (rands', midground) = makeDefMidground length rules rands seed
```

where *length* is the length in measures, *rules* is the set of production rules, *rands* is an infinite list of random numbers, and *seed* is the initial Roman numeral string. To produce the midgrounds with only one random number seed, the unused portion of *rands* is returned as *rands'*, which can be supplied to the next call of the function.

Because mapping to concrete notes was considered as a separate step for this implementation, a similar interface was used:

```
> (rands', midground') = toSimpleNotes rands midground
```

This step is responsible for choosing scale indices for each chord. These scale indices are then later converted to concrete notes using the tonic reference note supplied by the user.

6.2.1 Production Rules Used

The production rules presented here were largely derived from examples given in Renwick's "Analyzing Fugue: a Schenkerian Approach" [17]. Although the examples were using fugues, which are a more restrictive form, the main additional

restrictions are at the foreground level. Harmonies that would be acceptable in a classical fugue would also be acceptable in other contrapuntal music from the same time period. The set of production rules was further refined by trial and error during implementation to achieve better results. The sets of production rules were small for II, III, IV, VI, and VII due to lack of applicable examples in Renwick's text [17]. Production rules developed and implemented were the following:

- Start symbol: I_t. For sections not starting in the tonic, this start symbol is wrapped with a modulation of the form Mod-c(I_t), but production occurs regardless of the extra modulation terminal.
- Rules for I
  1. ($p = 0.25$) $\mathrm{I}_t \rightarrow \mathrm{II}_{t/4}\ \mathrm{V}_{t/4}\ \mathrm{I}_{t/2}$
  2. ($p = 0.25$) $\mathrm{I}_t \rightarrow \mathrm{I}_{t/4}\ \mathrm{IV}_{t/4}\ \mathrm{V}_{t/4}\ \mathrm{I}_{t/4}$
  3. ($p = 0.20$) $\mathrm{I}_t \rightarrow \mathrm{V}_{t/2}\ \mathrm{I}_{t/2}$
  4. ($p = 0.05$) $\mathrm{I}_t \rightarrow \mathrm{I}_{t/4}\ \mathrm{II}_{t/4}\ \mathrm{V}_{t/4}\ \mathrm{I}_{t/4}$
  5. ($p = 0.25$) $\mathrm{I}_t \rightarrow \mathrm{I}_t$

  These rules force any progression to end on the tonic. Enforcing that progressions end on the tonic provides more closure and usually sounds more final.
- Rules for II
  1. ($p = 0.80$) $\mathrm{II}_t \rightarrow \mathrm{II}_t$
  2. ($p = 0.20$) $\mathrm{II}_t \rightarrow$ Mod-VI($\mathrm{Vc}_{t/2}\ \mathrm{I}_{t/2}$)

  Here, since modulation to the relative second is not as common directly from the tonic as modulations to the subdominant/dominant in early classical music, an alternative modulation is allowable. In minor keys, this provides an alternative to a diminished chord. In major keys, it provides a modulation to the relative minor. An extra terminal is used in the modulation to avoid too abrupt a transition.
- Rules for III
  1. ($p = 0.80$) $\mathrm{III}_t \rightarrow \mathrm{III}_t$
  2. ($p = 0.20$) $\mathrm{III}_t \rightarrow$ Mod-III($\mathrm{IIIc}_{t/8}\ \mathrm{IIc}_{t/8}\ \mathrm{Ic}_{t/8}\ \mathrm{I}_{t/4}\ \mathrm{IIIc}_{t/8}\ \mathrm{Ic}_{t/8}$)

  The rules here are similar to those for II, except that modulation to the relative 3^{rd} is reasonably common, although modulations to the subdominant and dominant are more common.
- Rules for IV
  1. ($p = 0.50$) $\mathrm{IV}_t \rightarrow \mathrm{IV}_t$
  2. ($p = 0.50$) $\mathrm{IV}_t \rightarrow$ Mod-IV($\mathrm{Ic}_{t/4}\ \mathrm{Vc}_{t/4}\ \mathrm{I}_{t/4}\ \mathrm{Ic}_{t/4}$)

  The probabilities for modulation are increased for IV (compared to those for II and III), since modulation to the subdominant is common. The modulation production rule uses terminals to start and finish to enforce a smooth transition in and out of the modulation. These additional terminals were added to resolve observed problems in earlier implementations rather than based on a specific reduction rule example. The relative durations of the terminals for this rule are based on the assumption that sections are not terribly long (otherwise, excessively long sections of only a single terminal could result if the rule is used too early).

- Rules for V
  1. $(p = 0.15)$ $\mathrm{V}_t \rightarrow \mathrm{IV}_{t/2}\ \mathrm{V}_{t/2}$
  2. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \mathrm{III}_{t/2}\ \mathrm{VI}_{t/2}$
  3. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \mathrm{IV}_{t/8}\ \mathrm{VII}_{t/8}\ \mathrm{III}_{t/8}\ \mathrm{VI}_{t/8}\ \mathrm{V}_{t/2}$
  4. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \mathrm{V}_{t/4}\ \mathrm{VI}_{t/4}\ \mathrm{VII}_{t/4}\ \mathrm{V}_{t/4}$
  5. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \mathrm{V}_{t/2}\ \mathrm{VI}_{t/2}$
  6. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \mathrm{III}_t$
  7. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \text{Mod-VII}(\mathrm{I}_t)$
  8. $(p = 0.05)$ $\mathrm{V}_t \rightarrow \mathrm{VII}_t$
  9. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \mathrm{V}_t$
  10. $(p = 0.10)$ $\mathrm{V}_t \rightarrow \text{Mod-V}(\mathrm{IIc}_{t/8}\ \mathrm{Vc}_{t/8}\ \mathrm{I}_{t/8}\ \mathrm{IIc}_{t/8}\ \mathrm{Vc}_{t/4}\ \mathrm{Ic}_{t/4})$

  In Schenkerian analysis, V is a more general chord than some of the others. Therefore, V was given more possible production rules. Some rules were derived from Renwick's text [17], and others were experimental (the longest rule, for example) and were added to test limitations of the grammar.

- Rules for VI
  1. $(p = 0.50)$ $\mathrm{VI}_t \rightarrow \mathrm{VI}_t$
  2. $(p = 0.50)$ $\mathrm{VI}_t \rightarrow \text{Mod-VI}(\mathrm{Ic}_{t/4}\ \mathrm{I}_{t/4}\ \mathrm{Ic}_{t/4}\ \mathrm{IIc}_{t/4})$

  The modulation option was wrapped with terminals to ensure a reasonable entry and exit from the modulation.

- Rules for VII
  1. $(p = 0.50)$ $\mathrm{VII}_t \rightarrow \mathrm{I}_{t/2}\ \mathrm{III}_{t/2}$
  2. $(p = 0.50)$ VII_t

  Substitution of the I-III series was included due to the scale similarity between the relative third and seventh for both major and minor tonic scales.

These rules and their corresponding application probabilities are called *implRules* in the rules.lhs source file. Other alternative rule sets are presented, but the one described above was found to produce better results.

The approach of having production rules determine chord duration as a percentage of the original chord's duration was used because it easily reverses the process of reducing sets of short chords into single, longer chords. The exact duration values were based on the assumption that the meter is 4/4. It is assumed that, for a long chord's duration, t, αt will be an acceptable duration for a shorter chord in a production rule as long as α =2, 4, 8, etc. One downside to this approach is that it does not allow for constant durations to be assigned (e.g. a quarter note). This would particularly be important for rules that use terminals, since if applied too early, chords could be given durations that are too long. For a more general implementation, the ability to consider constant durations in addition to those of the form αt would be needed. However, this simple implementation, restricting rules to durations of the form αt did not yield undesirable results in most cases for the target genre. The choices of α values for these production rules also appeared to be the main factor contributing to the rhythms observed in the results, since the overall rhythmic structure across all voices was not altered much by foreground generation.

6.2.2 Algorithm for Grammar Application

An iterative algorithm was chosen because midground analysis is an iterative process. A probabilistic approach to choosing when to apply a production rule and which production rule to apply provided the most natural-sounding results. The number of iterations affects the probability that a non-terminal will eventually have a production rule applied, and as a result the expected length of each chord is dependent on the number of iterations. If each production rule application is recorded, then the derivation is preserved, then the Roman numeral portion of the corresponding analysis (excluding other properties of the voice-leading matrix as already described) is preserved. This algorithm does not allow back-tracking and gives equal consideration to each non-terminal from left to right for each iteration.

Algorithm 2 *Midground Grammar Application Algorithm.*

- *Input:*
  - *A minimum duration for chord type terminals and non-terminals, t_{min}, at or below which no production rules may be applied.*
  - *The probability p of applying a production rule to a non-terminal.*
  - *A starting string, ϕ, with duration t (usually $\phi = I_t$).*
  - *Maximum number of iterations: the number of times to pass over the symbol string from left to right.*
- *Procedure:*
  1. *For each iteration, for every non-terminal $C_{t'}$ in ϕ from left to right:*
     (a) *Find set R of all production rules with the form $C \rightarrow \phi'$.*
     (b) *With probability $(1 - p)$ leave $C_{t'}$ unchanged, otherwise: if $t' \leq t_{min}$ then leave $C_{t'}$ unchanged, otherwise choose $r \in R$ according to the probability distribution for R and replace $C_{t'}$ in ϕ with ϕ'.*
  2. *If any non-terminals remain in ϕ, perform the substitution $C \Rightarrow c$ to convert non-terminals to their corresponding chord type terminals.*

6.2.3 Mapping Roman Numerals to Concrete Notes

Initially, simple algorithms were explored for this purpose, including simple table lookup methods. However, these simple approaches demonstrated the following persistent problems: too monotone a sound (not enough exploration of range), getting "stuck" in excessively low/high ranges, an uneven distribution of chord inversions, and unnaturally large jumps in range. It was observed that good chord progressions tended to have a naturally rising and falling pattern over time, which started to sound melodic. The following algorithm was developed to try to accomplish this kind of shape for a progression. Table lookup is used to place chords at first, and then the progression is "shaped" from left to right in a single pass to try to add variation in the inversions present, provide a reasonable range, and avoid large jumps within a progression.

The initial assignment of indices uses the default indices in the sorted order given by table 1. I maps to indices [0,2,4], II maps to [1,3,5], etc. The triads occur in sorted order, such that the lowest voice will always occur first in the list. For a triad with indices [i_1, i_2, i_3], rotating the chord up converts it to the indices [i_2, i_3,

$i_1 + 7]$, and rotating the chord down transforms it to $[i_3 - 7,\ i_1,\ i_2]$. Indices < 0 occur in lower octaves, and those ≥ 7 are in higher octaves.

Excessive range is avoided by making it progressively less likely that chords will continue to be rotated in the same direction. The progression is allowed to wander to some degree, but will tend to come back to the range of the first chord in the progression periodically. The algorithm prevents progressions from becoming "stuck" in a range by making it progressively more likely that subsequent chords will be rotated in some direction when a series of chords are left unchanged.

The algorithm does not consider the spacing of the voices, although that would need to be considered in future implementations. Considering the voices as triads only leads to the voices being closely packed together and usually going up/down at the same time. Handling of voice spacing and whether the voices move in different directions is currently left to foreground generation.

Algorithm 3 *Brute-force Chords-to-Notes Algorithm.*

- *Inputs: a string of terminals, ϕ, a reference note n, and a mode m (e.g. major or minor), initial inversion probability p_i, and initial direction d_i (e.g. up or down).*
- *Procedure:*
  1. *Define a scale using the reference note and the mode. Index 0 in this scale will be the reference note. Indexes above and below 0 refer to notes within the scale.*
  2. *For each chord type terminal in ϕ, assign a default set of notes from a table such that every chord of the same type maps to the same initial notes. Chord types will be replaced with lists of scale indices. A list will still count as a single symbol in the string.*
  3. *Let current inversion probability $p_c = p_i$ and the current direction $d_c = d_i$.*
  4. *Set boolean flag unchanged = True.*
  5. *For each set of symbols, s in ϕ from left to right:*
     (a) *If s = Mod-c(...), then modify reference note and mode to be the relative c of the current reference note and mode for processing of terminals within the modulation. Once symbols inside the modulation are processed, restore the reference note and mode to their values prior to entering the modulation.*
     (b) *If s is a Term containing a list of indices, then:*
        i. *With probability p_c, rotate the stacking of the indices in s in the direction d_c.*
        ii. *If s was rotated, then:*
            A. *Perform the same rotation on all subsequent chords (even within modulations and across them)*
            B. *Set $p_c = p_c/2$*
            C. *Set unchanged = False.*
        iii. *If s was left unchanged:*
            A. *Set $p_c = p_c \times 2$*
            B. *If unchanged is False then reverse d_c, otherwise leave it unchanged.*
            C. *Set unchanged = True.*

6.3 FG1: Foreground Generation Implementation for MG1

The foreground generation implementation described here is both very simplistic and genre-specific. Pairs of notes are considered for modification using one of the following options: add a passing tone, add a neighboring tone (or other local minimum/maximum), suspend one note across the duration of both, or do nothing. Because the placement of the notes within a measure was not considered, removal of notes/addition of rests was not allowed. Although this option was implemented, it tended to create a meter-less sound when used regularly. As a result, all of the voices will be playing at any given time.

When choosing a note to add or suspend, the restrictions given in figure 3 are considered. Notes are added only when they do not violate those restrictions. If there is no legal option, nothing is done and the algorithm moves on to consider other foreground modifications.

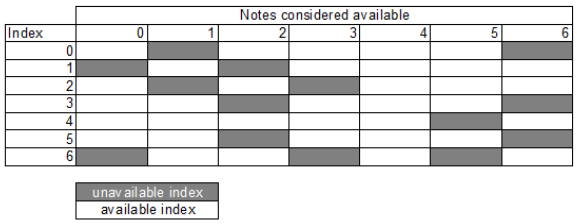

Figure 3: Table for acceptable note additions (columns), given another note being played at the same time (rows). The goal was avoidance of most dissonant intervals, such as minor seconds and tritones. The table's contents were determined partly by trial and error during experimentation. Although this particular table may not be optimal, it produced a better variety of single-line melodies than more restrictive tables that were tested and also resulted in fewer between-voice dissonances than less restrictive tables that were tested.

6.3.1 Algorithms for Individual Foreground Elements

Passing tones are notes that lead from one chordal tone to another. Since the output from the midground analysis implementation consists entirely of chordal tones in triads, adding passing tones can be accomplished by simply adding intermediate notes in the voices. The following algorithm was implemented to add a passing tone between two notes.

Algorithm 4 *Algorithm for adding a passing tone between notes n_1 and n_2 from the same voice:*

1. *If n_1 and n_2 are at the same pitch, do nothing, since a passing tone is impossible.*
2. *If n_1 and n_2 are at different pitches:*
   (a) *Let i_1 and i_2 be the scale indices of n_1 and n_2.*
   (b) *Choose a random index, i_r between i_1 and i_2.*

(c) *Let X be the set of all concurrent notes in other voices over the durations of n_1 and n_2.*

(d) *Let I be the intersection of the sets formed by looking up the valid indices for each member of X in the table given by figure 3.*

(e) *If I is the empty set, do nothing.*

(f) *If I has members, choose i_c as the member closest to index i_r and:*

i. *Let d_1 be the duration of n_1 and d_2 be the duration of n_2.*

ii. *With probability $1/2$: halve the duration of n_1 and insert a new note immediately after it with index i_c and $d_1/2$. Otherwise, halve the duration of n_2 and insert a new note immediately before it with index i_c and $d_2/2$*

A passing tone leading from one chordal tone to another must lie between the two tones and should not use a pitch that causes excessive dissonance. Notes held in the other voices over the duration of the two ending chordal tones must be considered when adding additional notes. Rhythm should also usually be considered, and this algorithm only takes a very rudimentary approach to it, assuming that divisions of note durations by 2 will be acceptable (which will be true if the input notes are quarter notes, half notes, etc.). It is possible that successive foreground changes may modify the chordal tones used as references when creating the passing tone, and that at a later time the new note may not actually constitute a passing tone anymore (it may become an anticipation, neighboring tone, etc.).

Neighboring tones are those that create a local maximum/minimum in a melodic phrase and settle on a chordal tone. A neighboring tone added between two notes should sit either above both notes or below both notes. Usually neighboring tones fall very close in pitch to the final chordal tone. To add diversity, the algorithm was implemented more loosely to consider a wider range of potential tones (up to an octave away). Although this means that not all notes added by this algorithm are neighboring tones in the strict sense, and others will simply be local minima/maxima leading to a chordal tone.

Algorithm 5 *Algorithm for adding a local minimum/maximum between notes n_1 and n_2 in the same voice:*

1. *Let i_m be the maximum (or minimum) of the indices of n_1 and n_2*
2. *Choose a random index, i_r between i_m and $i_m + 7$ (or $i_m - 7$).*
3. *Let X be the set of all concurrent notes in other voices over the durations of n_1 and n_2.*
4. *Let I be the intersection of the sets formed by looking up the valid indices for each member of X in the table given by figure 3.*
5. *If I is the empty set, do nothing. If I has members, choose i_c as the member closest to index i_r and:*

   (a) *Let d_1 be the duration of n_1 and d_2 be the duration of n_2.*

(b) *With probability* $1/2$*: halve the duration of* n_1 *and insert a new note immediately after it with index* i_c *and* $d_1/2$*. Otherwise, halve the duration of* n_2 *and insert a new note immediately before it with index* i_c *and* $d_2/2$

As with passing tones, these local minimum/maximum notes can only be added if they do not create excessive dissonance based on the table from figure 3. Like passing tones as well, it is also possible that further modification of other notes may cause new notes to no longer be local minimums/maximums in the melodic series in the final foreground.

Suspensions occur when a voice sustains a note across changing chords in other voices. Suspensions are common when chords share voices, but are also a way to introduce small amounts of desirable dissonance that are resolved later. Suspensions can be created in two ways by the following algorithm. Given two notes, only one is chosen and is sustained across the duration of both.

Algorithm 6 *Algorithm for adding a suspension over notes* n_1 *and* n_2 *in the same voice:*

1. *Let* X *be the set of all concurrent notes in other voices over the durations of* n_1 *and* n_2*.*
2. *Let* i_1 *and* i_2 *be the scale indices of* n_1 *and* n_2*.*
3. *Let* d_1 *and* d_2 *be the durations of* n_1 *and* n_2*.*
4. *Let* I *be the intersection of the sets formed by looking up the valid indices for each member of* X *in the table given by figure 3.*
5. *If suspending the first note and* $i_1 \notin I$ *is the empty set, do nothing. If suspending the second note and* $i_2 \notin I$*, do nothing.*
6. *If suspending the first note and* $i_1 \in I$*, then set* $d_1 = d_1 + d_2$ *and remove* n_2*. If suspending the second note and* $i_2 \in I$*, then set* $d_2 = d_1 + d_2$ *and remove* n_1*.*

Initial experimentation with foreground algorithms showed that suspensions that did not consider context were likely to produce potentially very bad dissonance rather than brief harmonic tension. As a result, a suspension will not occur if the note being suspended would be forbidden by the table in figure 3.

Anticipations are another way to add tension that resolves. An anticipation is created by playing a note before it occurs as a chordal tone.

Algorithm 7 *Algorithm for adding an anticipation:*

- *Split* n_1 *into two notes of equal duration. The first will have* n_1 *s pitch, and the second will have* n_2*'s pitch.*

Because this algorithm will resolve any dissonances immediately if applied directly to output from midground generation (rather than being applied to notes already modified by earlier foreground generation), consideration of context in the other voices was not included.

6.3.2 One-Pass Foreground Algorithm

In this implementation, foreground elements are added left to right in one pass in each voice in order. Once a foreground is added to one voice, that voice's foreground becomes part of the context for the next voice. This allows more freedom in the first voice concerning addition of new notes than in subsequent voices.

To aid in retaining good harmony, notes are still considered as scale indices within the foreground. The following data type was defined to handle this:

```
> data VTerm = VNote (Int, Dur) | VRest (Dur) | VMod (ModType, [VTerm])
```

A foreground for a voice is then a list of VTerm types. The algorithm for adding foreground elements is the following:

Algorithm 8 *One-Pass Foreground Algorithm.*

- *Given:*
  - *A list of VTerms, v_1, v_2, ..., v_n, for each adjacent pair of VTerms, v_i and v_j from left to right.*
  - *A list of foreground operations and probabilities of applying each.*
- *For each pair of adjacent VTerms, v_i and v_j:*
  1. *If either v_i or v_j is of the form $VMod(x, v_{k1}v_{k2}...v_{km})$, recursively apply the algorithm to $v_{k1}v_{k2}...v_{km}$ with a modulated scale and then continue with any VTerms following the VMod. Otherwise, continue.*
  2. *Choose a foreground operation to apply based on their application probabilities (see table 4).*
  3. *Replace v_i and v_j with a new string of VTerms, s based on the foreground algorithm.*
  4. *Only the last symbol in s may be reconsidered as v_i in the next step.*

When only a single pass of the algorithm is used, notes faster than an eighth note will not occur, and any give note will only be a part of two foreground considerations. Although the algorithm could be applied iteratively, this was found to quickly produce a meter-less sound and obscure the reasonable rhythmic patterns formed during midground generation.

This algorithm is not ideal, and could be improved in many ways. It would be ideal to operate on a chromatic scale and consider note compatibility based on mode, a change that was implemented in a modified version of foreground generation for

chord spaces. Although not implemented in either version of foreground generation, adding consideration for notes' rhythmic roles would, perhaps, also enable iterative use of the algorithm rather than only allowing it to run once on each voice. Note removal would also be a useful feature, assuming rhythm is taken into consideration. Functions for this were implemented but not used, since removing notes without considering rhythm made the results sound more random.

Two final manual additions to foreground generation were considered to alleviate problems observed by the algorithms observed so far:

1. The voices were too tightly clustered due to initial assignment as triads within a single scale. If the middle voice is transposed down an octave (becoming the lowermost voice), the voice spacing is improved.
2. The ending notes often did not sound final. The last chordal notes prior to running FG1 are repeated at the end of the composition and held for 2 measures. This is a common tactic used to emphasize an ending and also avoids ending on a dissonant chord due to alterations made in the foreground.

6.4 MG2: Midground Generation Implementation using Chord Spaces

The method for midground generation described in this section will be referred to as MG1. As an alternative to the algorithm presented in the previous section, the OPTIC chord spaces presented by [3] were implemented in Haskell. The first part of the algorithm is the same as in MG1, using the same grammar to produce a string of Roman numerals that can include modulations. In MG2, the process differs after this point, using chord spaces to turn the problem of mapping chords to notes into a path-finding or neighbor-finding problem. First, pitch space must be generated and converted into the appropriate equivalence classes. Then, matching equivalence classes are found for the Roman numerals and points within the classes must be selected as concrete notes. This approach requires that pitch numbers are used rather than scale indices.

6.4.1 Generating Pitch Space

Pitch space is generated as a list of all points for n voices where all voices are within the range $[p_{min}, p_{max}]$. The values of p_{min} and p_{max} are defined as constants in *Constants.lhs*. For this experiment, these constants were set to $p_{min} = -10$ and $p_{max} = 30$. Pitch space can be generated for n points and will initially consist of $(p_{max} - p_{min})^n$ points if no additional range constraints are imposed. When $n = 3$, there are 27,000 points. This set of points can then grouped into equivalence classes. Alternatively, since tone clusters such as $\{C, C\#, D\}$ are generally uninteresting from the standpoint of the types of progressions desired at this stage (for contrapuntal classical music), pitch space can be generated more compactly. As long as pitch space consists only of triads that are OPT-equivalent to major, minor, and diminished triads in root position, only potentially relevant points can be generated and then grouped into equivalence classes. Complete pitch space can also be reduced to the same number of relevant equivalence classes by applying OP-equivalence and then using OPT equivalence as a method to retain relevant equivalence classes.

The following representation was chosen for equivalence classes:

```
> type CLabel = AbsChord
> type EqClass  = (CLabel, AbsChordSet)
> type EqSet = [EqClass]
```

Equivalence classes are represented as a pair, (l, s), where s is the set or list of all points that belong to the class and $l \in s$. Pitch space can be initially be converted to a collection of equivalence classes where each point, x, is in the equivalence class represented by $(x, \{x\})$. The type *EqSet* was used to represent a collection of equivalence classes. The function *applyEqOp* was implemented to group equivalence classes under new relations.

```
> applyEqOp :: (AbsChord -> AbsChord -> Bool) -> EqSet -> EqSet
```

The OPT equivalence tests can be individually defined for use with *applyEqOp*, each having the type $AbsChord \rightarrow AbsChord \rightarrow Bool$.

```
> isOeq a b = normO a == normO b
> isPeq a b = sort a == sort b
> isTeq a b = normT a == normT b
```

where the functions $normT$ and $normO$ serve as ways to normalize chords to a standard representation for O and T respectively.

```
> normT c = map (\x -> x - c !! 0) c
> normO = map (`mod` 12)
```

Similarly, checks for OP and OPT-equivalence can be implemented in the way described in the approach sections:

```
> isOPeq :: AbsChord -> AbsChord -> Bool
> isOPeq a b = (sort.normO) a == (sort.normO) b

> isOPTeq :: AbsChord -> AbsChord -> Bool
> isOPTeq v1 v2 =
>     let v1' = (normT.sort.normO) v1
>         v2' = (normT.sort.normO) v2
>         s = map (normT.sort) (genOctStackings v2 0)
>     in  mapOr (map (== v1') s)
```

where the function *genOctStackings* creates all stackings of a particular set of pitch classes.

To obtain a set of equivalence classes where each Roman numeral has a single corresponding equivalence class, the O, P, and T relations can be used. Each of these equivalence classes will contain both a label that corresponds to a Roman numeral and a list of chords that meet the requirements for the corresponding Roman numeral.

First pitch space for $n = 3$ voices must be grouped into OP-equivalence classes, resulting in 364 equivalence classes if pitch space originally contains all possible combinations of pitches (as opposed to containing only collections of pitches meeting certain criteria, such as voice range restrictions). The relevant 36 classes can be extracted by preserving only those classes that are OPT-equivalent to the following:

- $[0, 4, 7]$, representing all major triads.
- $[0, 3, 7]$, representing all minor triads.
- $[0, 3, 6]$, representing all diminished triads.

This will find 12 major equivalence classes, 12 minor equivalence classes, and 12 diminished equivalence classes. Each one is potentially representative of a Roman numeral depending on the numeral's reference key. While $[0, 4, 7]$ will represent I in the home key of the progression (assuming transposition to the correct key after the midground is generated), $[2, 7, 11]$ represents Mod-V(I), since it takes into account the local transposition imposed by the modulation.

6.4.2 Forming Equivalence Classes under Multiple Relations

Two methods were examined for forming equivalence classes under multiple relations. The first version used a very brute-force approach and assumed that for some equivalence class, $e = (l, s)$, the points contained in s represent all of pitch space that belong to the equivalence class. Given two equivalence classes $e_1 = (l_1, s_1)$ and $e_2 = (l_2, s_2)$ formed under the equivalence relation R_1, similarity under another relation R_2 can be defined as follows:

$e_1 \sim_{R_2} e_2$ if $\exists x \in s_2.l_2 \sim_{R_2} x$ or $\exists x' \in s_1.l_1 \sim_{R_2} x'$

This allows use of individual equivalence tests in series. For equivalence relations R_1 and R_2, each can be applied independently to find the equivalence class structure under the combined relation R_1R_2. In this case, *isOeq*, *isPeq*, and *isTeq* can be used to successfully group pitch space using OP or OPT-equivalence. If *pspace* is all of pitch space as equivalence classes where each point occupies its own class, OP and OPT-equivalence can be found by:

```
op = ((applyEqOp isPeq).(applyEqOp isOeq)) pspace
opt = ((applyEqOp isTeq).(applyEqOp isPeq).(applyEqOp isOeq)) pspace
```

However, because this approach suffers from two significant problems:

1. It involves a lot of comparisons when searching for the correct z for $x \sim_{R_1} z \sim_{R_2} y$. This is potentially a very slow process when the sets of points to search are large.
2. If constraints are imposed on pitch space, an equivalence class, $e_1 = (l_1, s_1)$, may not have sufficiently many points in s_2 to be compared against another equivalence class, $e_2 = (l_2, s_2)$. If pitch space is restricted to the range $[0, 7]$ for all voices, the necessary intermediates for comparing $[0, 5, 7]$ and $[0, 2, 7]$ under OPT equivalence will not be present in either s_1 or s_2.

Because of these problems, under the assumption that shortcuts like *isOPeq* and *isOPTeq* exist, equivalence classes instead needed to be checked for equality under multiple equivalence relations by comparison of the labels only. This method allows OP and OPT-equivalence to be applied in one step:

```
op = ((applyEqOp isOPeq).toEqSet) pspace
opt = ((applyEqOp isOPTeq).toEqSet) pspace
```

6.4.3 Restricting Voice Ranges

Chord spaces represent a way to easily factor in instruments' range limitations. One pitch space has been grouped into equivalence classes of the form $e = (l, s)$, the set s can have elements retained or removed based on whether they violate range limitations for the voices. With this version of pitch space, the algorithm using the chord space does not need to factor in any additional range information when choosing voice-leadings.

Alternatively, pitch space itself can be generated along restricted ranges. This will improve runtime by eliminating unnecessary points, but it requires that only labels be checked when determining similarity of two equivalence classes. For a given equivalence class, $e = (l, s)$, the set s may not contain points needed for applying further equivalence relations with individual checks (such as *isOeq*, *isPeq*, and *isTeq* instead of *isOPTeq*).

For this implementation, voices were restricted to the following ranges by restricting pitch space:

- Highest voice: [12, 24]
- Middle voice: [12,24]
- Lowest voice: [-10,20]

The exact bounds on each voice are arbitrary, but the general trend in range was intended to allow two treble voices (mapped to Oboe and Clarinet) and one bass voice (mapped to Bassoon). Negative ranges were included for the lowest voice to allow for range differences between the tonic and dominant sections. After mapping to notes within the ranges above, all voices in the tonic were transposed up 4 octaves, and the dominant section was transposed up an additional 7 halfsteps to place it in the right key (the dominant).

6.4.4 Mapping Numerals to Points in Pitch Space

This takes place in three steps in order to reuse code from the previously described method for midground generation. For the first step, Roman numerals are mapped to the default assignment of scale indices. In the second step, using mode/scale type and modulation information, the scale indices for modes are converted to chromatic scale indices. In a major key, $[0, 2, 4]$ will be converted to $[0, 4, 7]$, and so forth. The datatypes needed for this step are the following:

```
> type RelativeChord = ([Int], Dur)
> data Term = Term RelativeChord | ModTerm (ModType, [Term])
>  deriving (Show, Eq, Ord)
```

These datatypes are reused from MG1, where each *RelativeChord* consists of a list of scale indices (such that $[0, 2, 4]$ is a I-chord). When converting from scale indices to concrete notes, the points will map to the fundamental domain for OP-space, with local applications of T to remove the need for modulations. This conversion algorithm is summarized below.

Algorithm 9 *Conversion Algorithm. For every term t in the list under mode/scale type s:*

1. *If t is a $Term(r, d)$, then convert r to its representative OP-class based on the mode used.*
2. *If t is a $Mod(m, ts)$, then determine the new mode, s', implied by m and recursively call the conversion algorithm on a version of ts transposed by the amount indicated by m.*

Once the conversion function has been applied to a list of Terms, two properties hold:

1. There are no remaining modulations in the list.
2. Every chord is represented by a point in the fundamental domain of OP-space with T equivalence at the progression level. More specifically, since the implementation only considers major and minor scales, each chord will fall into one of the 36 equivalence OP equivalence classes for major, minor, and diminished chords.

Now that every chord is represented by a point in the fundamental domain of OP-space, they can be mapped to specific points inside the same equivalence classes to add variation. For each term from left to right, a function called *chooseNextTermOP* was defined. It is called with the following arguments:

$$chooseNextTermOP\ t_{prev}\ t_{curr}\ r$$

where t_{curr} is the term being operated on and t_{prev} is a term that has already been modified. This function's behavior is described in algorithm 9.

Algorithm 10 *Algorithm for Finding Neighbors in Chord Spaces.*

1. *For chord arguments t_{prev}, t_{curr}, and a random number r, find S = the set of all points belonging to t_{curr}'s equivalence class.*
2. *Find $S' = all\ s \in S$ where the distance between s and t_{prev} is less than some threshold.*
3. *If there are no neighbors within the distance threshold, set $S' = t_prev$'s nearest neighbor in S.*
4. *Find $r_1 = r\ mod\ |S'|$*
5. *Return the r_1^{th} element of S'.*

This method can also help avoid large jumps in range between sequential sections with independently-generated strings of Roman numerals. For two sequential sections, A and B, A can be mapped to concrete notes first. The last chord from A can then be supplied as t_{prev} when mapping the first chord from B. This will give reasonable leadings between sections that are non-repeating, although jumps in range may still occur in scenarios such as $A - B - A$ form between B and the second occurrence of A.

Because the task of mapping chords to notes is now a path-finding or neighbor-finding problem, the distance measure and distance threshold used become quite important. Both are user-defined in this implementation, and can be modified by

changing the value of two global constants, *distMeasure* and *distThreshold* (defined in *Constants.lhs*). Four distance measures were explored. For two vectors $v_1 = [a_1, ..., a_n]$ and $v_2 = [b_1, ..., b_n]$:

1. Simple pitch distance = $|a_1 - b_1| + ... + |a_n - b_n|$

2. Euclidean distance = $\sqrt{(a_1 - b_1)^2 + ... + (a_n - b_n)^2}$

3. Maximum voice distance = $\max [|a_1 - b_1|, ..., |a_n - b_n|]$

4. Voice distance threshold = if ($|a_1 - b_1| \leq t_1 \wedge ... \wedge |a_n - b_n| \leq t_n$ then 0 else *distThreshold*+1, where each t_i is a voice-specific threshold.

Simple pitch distance and Euclidean distance are both common measures used when comparing chords. From the a voice-leading standpoint, choosing neighbors based on these measures is potentially problematic. The distance measure will have to be set reasonably high to allow all of the voices to move a moderate amount. For example, it may be acceptable for every voice to move up to 5 halfsteps in some direction. However, for that to be possible, the threshold would have to be set to 15. Using simple pitch distance and euclidean distance, this allows the possibility of the distance being due to movement in only one voice. If the threshold is set to 15 halfsteps, both simple pitch distance and Euclidean distance will allow a single voice to jump over an octave - which is undesirable in the style of music considered for this implementation. The maximum voice distance was found to be a more successful distance measure, since it allows each voice to move by a maximum amount. This avoids large jumps in any once voice while allowing reasonable freedom among the other voices. After experimentation with this measure under different distance thresholds, a threshold of 8 halfsteps was found to yield the most reasonable results.

Because the distance measure was only used to locate collections of neighbors, the voice distance threshold was used as an experimental alternative that gives different consideration to each voice. This function does not provide an actual distance measure between two chords, but rather gives a boolean response equivalent to labeling a chord as "close" or "far away." For "close" chords, the distance for each voice in the pair of chords falls within a voice-specific threshold. This allows preference to be given to different voices for smoother voice-leadings. Although this measure performed as expected and gave preference to smooth voice-leadings for voices with lower thresholds, the maximum voice distance measure was found to be more suitable to the target genre. Because of this, maximum voice distance was the measure selected to test algorithm robustness by generating collections of compositions.

6.5 FG2: Foreground Generation Implementation for MG2

Because MG2 produced output using pitch numbers rather than scale indices, the implementation for FG1 had to be adapted to suite the new representation. Foreground generation is most easily handled while still applying T-equivalence at the

progression level. The same algorithms for adding most melodic elements were retained unchanged from FG1, with a few differences:

1. Context-sensitivity was included for all functions responsible for adding foreground elements, with the exception of anticipations.
2. The algorithm for local minima/maxima were modified in two ways:
   (a) Both algorithms from FG1 were changed to select a new pitch class in the same way, but then the placement of the pitch class was dictated by the nearest placement. As a result, notes created may be local minima, local maxima, or in some cases even passing tones.
   (b) For selecting a new pitch class to introduce, both available indices using the note availability tables shown in figures 4 and 5 and any pitch class being played in the immediate context are possibilities. This change was made to compensate for the relative strictness of the note availability tables compared to those from FG1.
3. Duplicate notes are combined in an additional probabilistic pass over each voice. Duplicate notes are combined with probability 0.5 once all modifications have been made with the FG1's one-pass foreground algorithm.
4. The tables for choosing foreground notes were adapted to work in pitch space rather than with scale indices. These are shown in figures 4 and 5.
5. Foreground operation probabilities (adding a passing tone, creating a suspension, etc.) were altered to slightly to improve observed melodies. This is shown in table 4.

Once foreground generation is complete, the result can be transposed into to the desired key. Only the second manual foreground addition from FG1 was used (repeating the last chord). Further spacing of the voices was unnecessary.

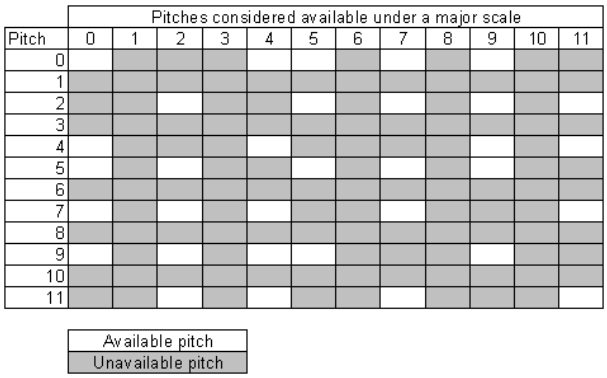

Figure 4: Table for acceptable note additions (columns), given another note being played at the same time (rows) in a major mode. This table is based on the table shown in figure 3.

6.6 Results from Generation Using MG1 and FG1

Both major and minor modes were tested. For the best comparison between results, all output was generated with a reference note of C5 for the tonic sections (A and

	Pitches considered available under a minor scale											
Pitch	0	1	2	3	4	5	6	7	8	9	10	11
0												
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												

Available pitch
Unavailable pitch

Figure 5: Table for acceptable note additions (columns), given another note being played at the same time (rows) in a minor mode. This table is based on the table shown in figure 3.

	Probability of Foreground Operation						
	PT	LMU	LML	ANT	SUS1	SUS2	NONE
FG1	0.2	0.2	0.2	0.2	0.05	0.05	0.1
FG2	0.25	0.2	0.2	0.2	0.05	0.05	0.05

PT = passing tone, ANT = anticipation, NONE = no change
LMU = local maximum (FG1) or local min/max using higher note as reference (FG2)
LMU = local minimum (FG1) or local min/max using lower note as reference (FG2)
SUS1 = suspension using first note
SUS2 = suspension using second note

Table 4: Comparison of probabilities for each foreground operation for FG1 and FG2.

A'). Generally, major and minor results sounded comparable for the same random number seeds. Minor results showed slightly more frequent usage of diminished chords, which was somewhat undesirable. However, this type of behavior would not be possible to correct without considering major and minor chords separately within the grammar. A II-V-I progression in a major tonic will have no diminished chords, while a II-V-I progression in a minor tonic will result in a diminished II-chord unless the scale is changed for that chord to allow a standard minor triad. This suggests that consideration of mode in the midground grammar would be a useful modification.

Despite the increased use of diminished chords in minor tests, the results were generally reasonable and met expectations, although they were usually not as good as those generated by the MG2 and FG2 implementations. Judging the quality of any music is a subjective process and although a quantitative analysis of the results was not possible, output from this implementation was examined for roughly how well it met the target genre and whether it had noticeable defects, using standard classical composition guidelines for contrapuntal music [2, 13]. Most of the results were roughly of the same quality, although some were slightly better than expected (namely TEST_15 and TEST_32). However, all results contained at least a small number of harmonies or voice leadings that would be considered flaws for the genre due to the simplicity of the foreground algorithm and lack of midground consideration for mode. Although the melodies produced in the voices were unique with each random number seed tried, the overall patterns demonstrated a lot of similarity. This meant that extremely bad results containing a lot of dissonance or poor

rhythm were unlikely, but it also meant that most results simply sounded average and that the algorithm was not very versatile.

The most common problems observed the following:

- Excessive use of diminished chords (II for a minor tonic, VII for a major tonic).
- Progressions traveling too high or low in range.
- Rough-sounding transitions where modulations occur and between sections (such as in figure 6), primarily unusual chord transitions and too large jumps in range.
- Modulations occurring early in midground generation, resulting in the sense of tonic being shifted to the wrong key.

Some of these problems, such as rough transitions between sections, may be corrected by introducing context-sensitivity into the midground grammar. Correction of the other algorithms would probably require algorithm modifications that consider Schenkerian analysis and other aspects of music theory in more detail.

Harmony and the chord progressions generated by midground algorithms were mostly acceptable for the intended genre, although almost all of the output contained a noticeable number of faults. Some unusual chord choices and chord transitions occurred, and foreground analysis would sometimes produce less-than-optimal harmonies. Undesirable harmonies were limited to odd chord transitions, inappropriate use of diminished chords (where a single-chord modulation probably should have been substituted instead), and allowing major seconds to occur between notes in the same octave (resulting in small tone clusters). For example, in the key of C-major (or C-minor), foreground analysis would deem F and G as suitable concurrent notes. However, when the notes are in different octaves the result usually sounds better than when the notes occur in the same octave, resulting in a major second.

Melodies within a single voice were also often acceptable, and sometimes sounded better than the result with all three voices combined. Probably because the highest voice was least constrained during foreground generation, this voice usually had the most pleasing melody of the three voices. This supports the Schenkerian harmony-centric view of this type of music, since the results from the implementation showed that a harmony-centric process could produce melodic results.

One of the more interesting features of the results was the fact that rhythm was reasonable, despite the fact that addition of rhythmic elements at the foreground level was largely random. Notes were split and combined without regard to their placement within measures or larger phrases, yet the time signature remained clear and there were few rhythmic anomalies that sounded out of place for the genre. Because the foreground in this implementation ignored rhythmic context, it suggests that midground played a significant role in creating acceptable rhythmic frameworks.

6.7 Results from Generation Using MG2 and FG2

The chord spaces implementation for midground generation yielded several improvements over the output observed from the implementation using MG1 and

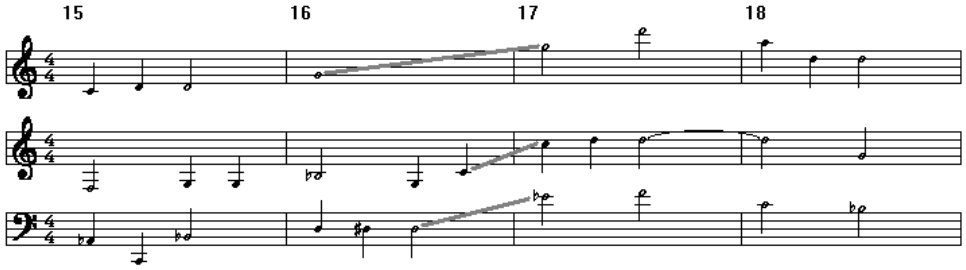

Figure 6: An example of an inappropriately large jump (indicated by the thick, gray lines) occurring in all voices in test output 11 from the implementation using MG1 and FG1. Part A' ends at measure 16, and part B begins at measure 17, indicating that the jump is due to the sections having been fully independently generated.

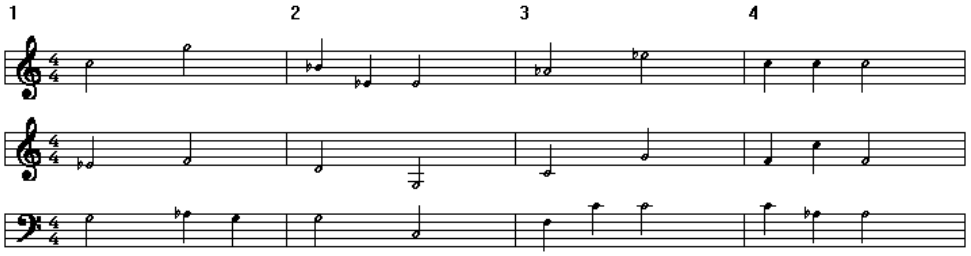

Figure 7: An example of more suitable melodies and voice leadings demonstrated by the opening measures of test output 4 from the implementation using MG1 and FG1.

FG1.

1. Once each numeral had been mapped to concrete notes in pitch space using the neighbor/path-finding approach, a reasonable distribution of chord stackings was present in the progression.
2. The amount of of similar and contrary motion in the voices both before and after foreground generation was slightly more in keeping with standard counterpoint guidelines. The implementation using MG1 and FG1 showed noticeably large amounts of similar and parallel motion, which is considered undesirable [2].
3. Voices rarely strayed into ranges that seemed inappropriately high or low. When this did occur, it was due to the addition of local minima or maxima by FG2. The output from MG2 stayed consistently within a reasonable range.
4. Noticeable/inappropriately large jumps (such as an octave or more) in range made by all voices simultaneously were uncommon compared to the results from MG1 and FG1. Some large jumps still appeared between sections, but large jumps in more than one voice simultaneously were rare.
5. The voices demonstrated better spacing with no additional transpositions to separate them.

Observations 1 and 2 are due to the use of a path-finding algorithm through pitch space (compared to the brute force approach from MG1). Observations 3-5

were due to the a combination of implementation of restricted voice ranges in pitch space and the distance measure used (maximum voice distance).

The results were generally more reasonable than those from the implementation using MG1 and FG1, although switching to use of MG2 and FG2 did not fix all of the problems and the results were again very uniform in sound. Persistent problems were the use of too many diminished chords (again due to lack of major/minor distinction in the grammar) and the occasional large jump added in the form of local minimums or maximums in the FG2 algorithm. The foregrounds produced by FG2 also still had periodic strange dissonances, and the melodies produced were sometimes not as good as those from FG1. The difference in melodic quality is attributable to the increased freedom in voice movement, and may be correctable with fine-tuning of the distance measure and distance threshold. Still, the results from MG2 and FG2 were more consistently consonant than those from MG1 and FG1 and demonstrate an overall more flexible approach to finding varied voice-leadings. To avoid the periodic dissonances and to further improve upon the melodies seen from the MG1 and FG1 implementations, the FG2 foreground algorithm would need to be more complex. For example, it would be useful to consider notes at more levels than simply whether they are legal or illegal to add.

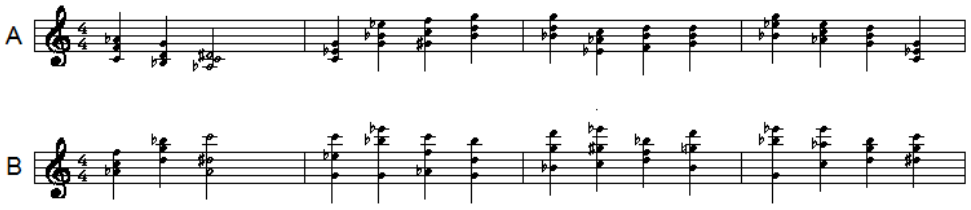

Figure 8: A comparison of chord progressions generated by the MG1 (example A) and the MG2 implementations (example B) using a minor mode and random number seed 0. Example A shows the tight clustering of the voices prior to transposition of the middle voice, as well as the fact that the voices exhibit mostly similar motion (a feature that will not change when transposing the middle voice). This behavior is improved by the algorithm in example B, where the voices move more independently of each other despite the fact that the pitch classes for each chord are exactly the same as those from example A. The duration of each chord was shortened by a factor of two in order to show more chords.

7 Conclusions and Future Work

The results from this reverse-Schenkerian approach to music generation made significant improvements over algorithms that use simple grammars alone [21] and algorithms that use chord spaces without consideration for standard forms of compositional development [12]. However, the range of results was very narrow and shows that this kind of approach is not very generalized. Although the algorithms presented here would be extensible in future implementations to different numbers of voices, the results would still be limited largely to the same genre. To create another style of music, new rule sets (and perhaps even a new grammar application

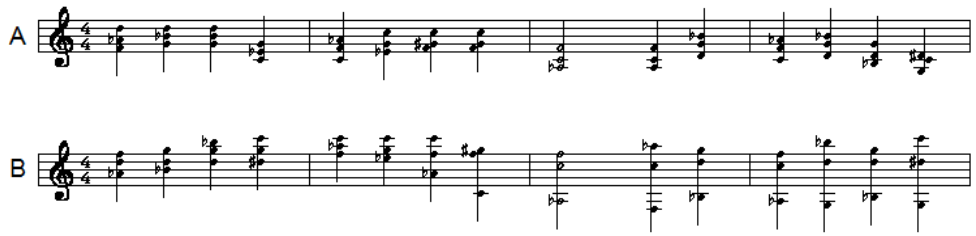

Figure 9: A comparison of chord progressions generated by the MG1 (example A) and the MG2 implementations (example B) using a minor mode and random number seed 1. The duration of each chord was shortened by a factor of two in order to show more chords.

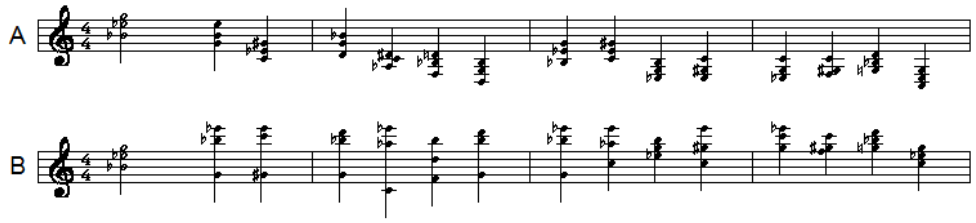

Figure 10: A comparison of chord progressions generated by the MG1 (example A) and the MG2 implementations (example B) using a minor mode and random number seed 2. The duration of each chord was shortened by a factor of two in order to show more chords.

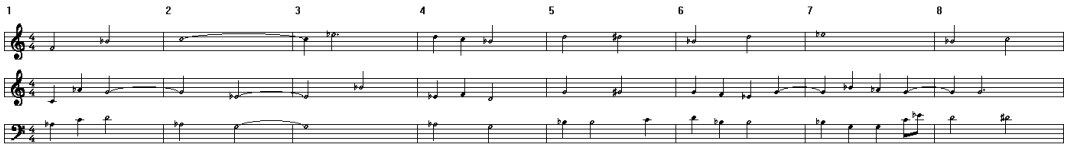

Figure 11: An example of output from the implementation using MG2 and FG2 using random number seed 0 and a minor tonic. The midground for the first four measures can be seen in figure 8. Because of the shortened durations in figure 8, a quarter note in figure 8 would map to a half note in the score above.

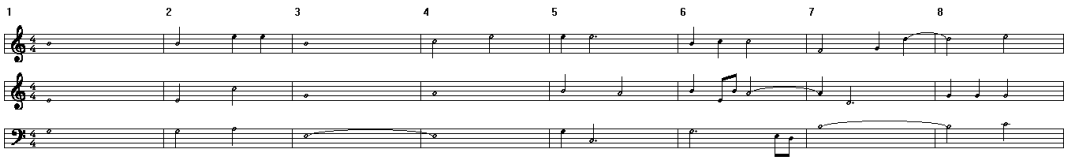

Figure 12: An example of output from the implementation using MG2 and FG2 using random number seed 6 and a major tonic.

algorithm) would be needed to create a midground, and new algorithms would need to be developed for creating foregrounds and handling rhythm.

7.1 Possible Grammar Additions

To consider other forms of music, the terminals and non-terminals would need to be diversified. For example, to make good jazz harmonies, not only would a major II-chord need to be differentiated from a minor II-chord (usually denoted as ii), other modifications of chords would probably also need to be addressed: notes in a 4th voice (7^{th}s and 9^{th}s, for example), sharp/flat notes that do not fall within the tonic mode, "sus" chords (those omitting the third), etc. Although the results for the implemented grammar did not show too much difference in quality between modes, diminished II-chords were evidence of the fact that modes need to be handled separately, even if they are not distinguished from an analysis standpoint. Some additional grammar modifications to give the grammar a finer degree of control over the development pattern would also be useful.

Another aspect of music that is not supported directly by the type of grammar described already for midground generation is the tendency to repeat sections verbatim, or to repeat sections with only slight variations at the foreground level. Although this was considered at the background level, it can also occur within smaller sections of music. To ensure that some number of identical non-terminals would be replaced with identical strings, it would require the addition of some sort of Let-in statement to define a different type of production behavior that is not typical of generative grammars. The following syntax would capture this behavior:

$$\text{let } x = \phi \text{ in } (\phi_2)$$

where ϕ_2 is a string that can use the symbol x to denote identical copies of some section of music. The statement $x = \phi$ means that x is result of running a grammar application algorithm with input ϕ is the initial seed of non-terminals and terminals. This result, x, and will be the exact same string for every occurrence of x in ϕ'. For example:

$$\text{let } x = \phi \text{ in } (x\ x\ \mathrm{I}_t)$$

If for some grammar application algorithm A, $A(\phi) \Rightarrow \phi'$ and $\mathrm{I}_t \Rightarrow \phi''$, then $x\ x\ \mathrm{I}_t \Rightarrow \phi'\phi'\phi''$. Depending on how the grammar application algorithm behaves, it may not be impossible that phi' and phi'' could be the same string. However, from a musical standpoint, it would only make sense for this to happen with very small probability, since the goal of defining the pattern "$x\ x$ I" would be to repeat a section twice followed by something new.

Additional production rules could then be added using let-in statements to expand non-terminals. For example:

$$\mathrm{I}_t \rightarrow \text{let } x = \mathrm{I}_{t1} \text{ in } (x\ x\ \mathrm{V}_{t2}\ x)$$

This kind of rule would allow for a finer degree of control over how much repetition and variation occurs within a section. However, it only controls these properties at the chord level. Foreground generation is a separate process and would not

necessarily have to produce the same melodies for duplicate chord progressions.

Unfortunately, although some initial implementation was put in place to support these let-in statements, they were not explored fully.

7.1.1 Combining Background and Midground Generation

With the ability to define repeating sections added to midground generation, background generation could simply be defined as a series of let-in statements. For example, a form like A-B-B-A could be defined as follows:

let $a = \phi_1$ in (let $b = \phi_2$ in $(abba)$)

Similarly, since the backgrounds for A-A-B-A and A-A'-B-A form would be identical in this system, both forms could also be defined this way:

let $a = \phi_1$ in (let $b = \phi_2$ in $(aaba)$)

However, in the case of A-A'-B-A, the assumption would be made that the foreground for A and A' would not be identical (since it is a separate process). If the background algorithm would need to stipulate the difference between A and A' (rather than leaving it up to the foreground generation), then these let-in style statements would require some additional information to be recorded about each section. An small implementation was created to support this kind of statement, but preference during experimentation was given to studying the effects of chord spaces and complete integration of let-in statements into the implementation was not pursued.

7.1.2 Proposed Future Work

The results presented here show promise, but there is still a lot of work to be done before a more generalized algorithmic composition method is possible. There are several ways in which the algorithms explored could be improved upon to make progress towards this goal.

The Schenkerian analysis-inspired portion of the implementation is one place that could be improved. Ultimately it may be useful to have a grammar that operates directly on a chord space to retain the idea of a voice-leading matrix. Other path-finding algorithms, such as Markov chains, may also be worth exploring as an alternative to the Schenkerian approach, since these would allow the possibility of learning the model from examples. Some machine learning approaches that address grammars in natural language may be useful, since recent research has shown that music and language are handled by strongly overlapping parts of the brain [11]. Machine learning has addressed the issue of learning semantics in addition to syntax [19], which might be useful to not only learn reasonable patterns in music, but also to differentiate different patterns based on the type of mood created.

Rhythm is also an aspect of music that warrants more attention during the generative process, or even its own generative process separate from harmony construction. Although the results from this implementation demonstrate that rhythmic tendencies can be built into a grammar at the mid-ground level, rhythm should also be a large part of foreground generation, particularly when considering styles of music that have very rhythmically complex melodies (jazz, for example). Leaving

rhythm as a side effect of harmony production rules would not be the most efficient or generalized way to handle rhythm in more complex music. Some approaches for analyzing rhythms, such as those by Paiement et al. [9], may yield useful generative models.

While Schenkerian analysis-inspired approach explored in this paper does not provide sufficient structure for rhythm by itself, other geometric representations that attempt to factor in rhythm may be useful. Some representations can model developmental patterns in music as graphs that capture temporal, and therefore rhythmic information in the edges used to connect nodes representing pitches. The resulting geometry of the graph is then directly related to the rhythmic structure of the music [23]. While these methods are primarily analytical, they may be possible to use generatively as well.

These kinds of models may also benefit from a more generalized notion of tonality. The tonnetz is a topic that has been explored as a means to relate similar-sounding chords at the pitch class level. The tonnetz may be viewed as a tiling [15, 24], and also as chords related by various geometric transformations [5]. This may yield either an alternative to chord spaces or a useful additional layer to combine with them.

A Supporting Chord Space Lemmas

This section provides supporting lemmas for the following, where pitch space is assumed to be $\mathbb{Z}^n$:

$\forall x, y \in \mathbb{Z}^n \forall f_1 ... f_n \in \{O, P, T\}$, if $\exists z_1, ..., z_{n-1} \in \mathbb{Z}^n.\ x \sim_{f_1} z_1 \ ... \ z_{n-1} \sim_{f_n} y$, then $\exists o_1 \in O, \exists p_1 \in P, \exists t_1 \in T.\ x \sim_{o_1} w_1 \sim_{p_1} w_2 \sim_{t_1} y$.

In the following discussion, it is assumed that pitch space is infinite for all voices. A sufficiently large subset of pitch space, S, for all voices can be substituted, as long as it is large enough to ensure that for any two points in the considered range, $x, y \in S$, and any two individual OPT relations, R_1 and R_2, if $x \sim_{R_1 R_2} y$ then there is a point, $z \in S$, such that $x \sim_{R_1} z \sim R_2 y$.

A.1 Haskell Implementations of the OPT Relations

The O, P, and T relations can be described as sets of functions, where the mapping of input x to output y represents $x \sim y$. Since the OPTIC relations are intended to be equivalence relations, the following must hold for all inputs x:

Reflexivity: $x \sim x$.
Symmetricity: if $x \sim y$ then $y \sim x$.
Transitivity: if $x \sim y$ and $y \sim z$ then $x \sim z$.

These functions are defined in Haskell to aid proving properties about them.

```
>type AbsChord = [AbsPitch]

> o :: [Int] -> AbsChord -> AbsChord
```

```
> o = zipWith (\i -> \p -> p + (12*i))
```

This function will take a list of octave shifts, each value indicating the number of octaves (which may be negative) to shift the corresponding value in the chord argument. It is assumed that the first and second arguments will have the same cardinality. For example:

$o\ [-2, 1, 0]\ [60, 61, 62] \mapsto [36, 73, 62]$

Each $o_i = o\ octs$, where $octs \in Z^n$ ($octs$ is an integer list), defines a function of type $AbsChord \rightarrow AbsChord$. This function represents a set of relations.

```
> p :: [Int] -> AbsChord -> AbsChord
> p [] chord = chord
> p (i:is) chord =
>     let index = length chord - length (i:is)
>         newC = p is chord
>         newC' = replaceIth index (chord !! i) newC
>     in  newC'
```

The p function is like the o function, except that it takes a list of permutation indices as its first argument. For example:

$p\ [2, 0, 1]\ [60, 61, 62] \mapsto [62, 60, 61]$

Each $p_i = p\ perm$, where $perm$ is a permutation, represents a set of relations. It is assumed that only valid permutations will be supplied to p. For this to be enforced, p would need some sort of wrapper function, such as the following.

```
> p2 :: [Int] -> AbsChord -> AbsChord
> p2 inds chord =
>     let lenCheck = length inds == length chord
>         m = (length chord) - 1
>         containsCheck = and (map (`isIn` inds) [0..m])
>         pass = containsCheck  && lenCheck
>     in  if pass then p inds chord
>         else error "(pWrap) Invalid index list."
```

With this added constraint, p will be a symmetric group: the set of all permutation functions on a set with composition as the group operator [6]. This gives rise to the following useful properties of P:

- $\forall p_1, p_2, p_3 \in P.\ (p_1 \cdot p_2) \cdot p_3 = p_1 \cdot (p_2 \cdot p_3)$. Associativity is a requirement for all groups [6].
- $\forall p_1, p_2 \in P,\ \exists p_3 \in P.\ p_1 \cdot p_2 = p_3$. This is derived from the property of groups in general [6].

- $\forall p_1, p_2 \in P, \exists p_3 \in P.\ p_1 \cdot p_2 = p_2 \cdot p_3$. Since the function $(p_1 \cdot p_2)$ defines a mapping from one ordering to another, p_3 imply needs to reorder the items to the same final state after obtaining the ordering from p_2. The symmetric group contains all possible permutation functions, so it must contain p_3.

```
> t :: AbsPitch -> AbsChord -> AbsChord
> t c = map (+c)
```

Each $t_i = t\ const$, where $const \in Z$, represents a set of relations. Proofs on these relations can proceed by using proofs on the functions, as for O.

A.2 Lemmas for OPT

The following lemmas are needed to prove the claim at the beginning of the appendix.

A.2.1 Commutativity Lemmas

Lemma 11 $\forall o_1, o_2 \in O.\ o_1 \cdot o_2 = o_2 \cdot o_1$

Proof. By commutativity of the + operator. For each pitch x_i in a chord and each corresponding octave transformation value o_{1i} and o_{2i}:

$$x_i + (o_{1i} \times 12) + (o_{2i} \times 12) = x_i + (o_{2i} \times 12) + (o_{1i} \times 12)$$

Or, restated:

$$(o\ octs_1) \cdot (o\ octs_2) = o\ (octs_1 + octs_2) = o\ (octs_2 + octs_1) = (o\ octs_2) + (o\ octs_1)$$

■

Lemma 12 $\forall t_1, t_2 \in T.\ t_1 \cdot t_2 = t_2 \cdot t_1$

Proof. By commutativity of the + operator.

$$(t\ c_1) \cdot (t\ c_2) = t\ (c_1 + c_2) = t\ (c_2 + c_1) = (t\ c_2) + (t\ c_1)$$ ■

A.2.2 Associativity Lemmas

Lemma 13 $\forall o_1, o_2, o_3 \in O\ (o_1 \cdot o_2) \cdot o_3 = o_1 \cdot (o_2 \cdot o_3)$

Proof. By commutativity and associativity of the + operator, since the o function performs vector addition.

$$\begin{aligned}
&(o\ octs_1) \cdot ((o\ octs_2) \cdot (o\ octs_3)) \\
&= (o\ octs_1) \cdot (o\ (octs_2 + octs_3)) \\
&= o\ (octs_1 + octs_2 + octs_3) \\
&= (o\ (octs_1 + octs_2)) \cdot (o\ octs_3) \\
&= ((o\ octs_1) \cdot (o\ octs_2)) \cdot (o\ octs_3)
\end{aligned}$$

■

Lemma 14 $\forall t_1, t_2, t_3 \in T\ (t_1 \cdot t_2) \cdot t_3 = t_1 \cdot (t_2 \cdot t_3)$

Proof. By commutativity and associativity of the + operator, since the t function adds a constant uniformly across a vector.

$(t\ c_1) \cdot ((t\ c_2) \cdot (t\ c_3))$
$= (t\ c_1) \cdot (t\ (c_2 + c_3))$
$= t\ (c_1 + c_2 + c_3)$
$= (t\ (c_1 + c_2)) \cdot (t\ c_3)$
$= ((t\ c_1) \cdot (t\ c_2)) \cdot (t\ c_3)$ ■

A.2.3 Transitivity

Substituting one function for a set of functions demonstrates transitivity. If F can be broken into a collection of functions representing sets of relations: $\exists f_1, f_2, f_3 \in F.\ f_1(a) = b, f_2(b) = c, f_3(a) = c$ implies $a \sim b$, $b \sim c$, and $a \sim c$ under the relation F.

Lemma 15 $\forall o_1, o_2 \in O, \exists o_3 \in O.\ o_1 \cdot o_2 = o_3$

Proof. By commutativity and associatitivty of the + operator. Given $o_1 = o\ octs_1$ and $o_2 = o\ octs_2$, we can define $o_3 = o\ (octs_1 + octs2)$. ■

Lemma 16 $\forall t_1, t_2 \in T, \exists t_3 \in T.\ t_1 \cdot t_2 = t_3$

Proof. By commutativity and associativity of the + operator. Given $t_1 = t\ c_1$ and $t_2 = t\ c_2$, we can define $t_3 = t\ (c_1 + c2)$. ■

A.2.4 Identity Functions

Lemma 17 $\exists o_{id} \in O, p_{id} \in P, t_{id} \in T.\ id = o_{id} = p_{id} = t_{id}$

Proof. The identity functions are:

$o_{id} = o\ [0, ..., 0]$
$p_{id} = p\ [0, 1, ..., n-1]$
$t_{id} = t\ 0$

This gives reflexivity for O, P, and T. ■

A.2.5 Reordering and Substitution

Lemmas are assumed to be for fixed values of n unless otherwise stated. Where C operations are involved, O^n and P^n refer to functions for specific numbers of voices (rather than referring to repeated application of the relation).

Lemma 18 $\forall o_1 \in O, p_1 \in P,\ \exists o_2 \in O.\ p_1 \cdot o_1 = o_2 \cdot p_1$. *More specifically, if* $o_1 = o\ octs$, *then* $o_2 = o(p\ octs)$.

Proof. We have that $p_1 \cdot o_1 \neq o_1 \cdot p_1$. This is because the octave shift arguments to o_1 would need to be permuted in order to apply octave equivalence after p_1 to achieve the same results as $p_1 \cdot o_1$. Therefore, given $o_1 = o\ octs$, we can define $o_2 = o\ (p_1 octs)$. ■

The following are examples of this property on triples of functions:

$$\begin{array}{rll} & o_1 \cdot p_1 \cdot o_2 & \text{Find } o_2' \text{ for reordering of } o_2 \text{ and } p_1. \\ = & o_1 \cdot o_2' \cdot p_1 & \text{Find } o_3 \text{ from combining } o_1 \text{ and } o_2'. \\ = & o_3 \cdot p_1 & \text{Final form.} \end{array}$$

$$\begin{array}{rll} & p_1 \cdot o_1 \cdot p_2 & \text{Find } o_1' \text{ for reordering of } p_2 \text{ and } o_1. \\ = & p_1 \cdot p_2 \cdot o_1' & \text{Apply symmetric group property to obtain } p_3'. \\ = & p_3' \cdot o_1' & \text{Find } o_2 \text{ for reordering.} \\ = & o_2 \cdot p_3 & \text{Final form.} \end{array}$$

Lemma 19 $\forall t_1 \in T, f \in O \cup P, t_1 \cdot f = f \cdot t_1$

Proof. Because t operates uniformly and because of associativity and commutativity of the + operator, it may be applied before or after any of the O or P functions without changing the overall function. ■

Lemma 20 $\forall f_1 ... f_n \in O \cup P \cup T,\ \exists o_1 \in O, \exists p_1 \in P, \exists t_1 \in T.\ f_1 ... f_n = o_1 \cdot p_1 \cdot t_1$

Proof. $f_1 ... f_n$ can be reordered to group all similar operations using the lemmas already described. Once grouped, operators can be combined using the substitution lemmas to have a series of exactly one of each type of function. Some of these functions may be the identity function (id is already a member of O, P, and T). ■

An algorithm that finds some series of operations $f_1, ..., f_n$ in OPT such that $x \sim_{f_1 \cdot ... \cdot f_n} y$ has successfully shown that $x \sim_{OPT} y$, although it may find more intermediates than are necessary. The minimum number of intermediates can be found by condensing the series of operations applications.

B Output and Source Code File Descriptions

All source code and output files can be found online at:

http://pantheon.yale.edu/~dvq2/691_files.zip

Output consists of MIDI files numbered based on the random number seed used to produce them. Each file is called TEST_#, where # is value supplied for *seed* in main.lhs. Output produced with the *mode* value in main.lhs set to major and minor for the same random number seeds are separated into folders labeled according to mode. Output from the MG1+FG1 implementations are in the folders called "major" and "minor," while output from the MG1+FG2 implementations are in the folders called "major_cs" and "minor_cs."

Output from MG1+FG1 Major output was produced for *seed* values of 0-19, and minor mode output was produced for values of 0-19, 32, 52, 78, 83, 105, 324, and 400. A sample performance using the "Phil" algorithm (developed for CPSC 542) is also given for $seed = 32$ in the file TEST32P.MID. The file TEST32Q.MID contains tempo changes (rubato) that were added by hand, and TEST32R.MID

transposes the B section down an octave into a more suitable range for the selected instruments. All other output files are unaltered from the output produced by main.lhs with the stipulated value for *seed*.

Output from MG2+FG2 Major and minor output were produced for *seed* values in the range of 0-19. A sample performance using the "Phil" algorithm is given for $seed = 6$ in a minor mode in the file TEST6P.MID. The file TEST6Q.MID contains tempo changes that were added by hand to TEST6P.MID.

Comparisons of MG1 and MG2 Output used for the figures showing chord progressions from MG1 and MG2 can be found in the "comparisons" folder. Output is divided by the mode used, and examples came from tests in the "minor" folder. Output files from the MG1 implementation start with "ptest" and those from the MG2 implementation start with "ptestc". The specific comparisons used in figures are contained in the files beginning with "comp."

B.1 File Descriptions

Source code was broken into different modules, each with its own file. These are described below. Most files were developed specifically for the CPSC 690/691 project, but some performance-related modules were developed for CPSC 542 (just listed as 542 in the table). Since the 690/691 code was used to produce compositions to test a performance algorithm for CPSC 542, the performance-related code is included.

Filename	Origin	Description
Constants.lhs	690/691	User-defined constants to be referenced by multiple other modules.
Foreground.lhs	690/691	Functions for adding melodic elements to a midground.
Foreground2.lhs	690/691	Adaptation of Foreground.lhs to chord space-based midgrounds.
GrammarTypes.lhs	690/691	Data types used by other modules for all steps of generation.
Instruments.lhs	542	Custom instrument definitions for performance.
Main-old.lhs	690/691	Defines the A-A'-B-A format composition and acts as a tutorial for the other modules. This version uses the original algorithms for midground generation and does not use chord spaces.
Main-optic.lhs	690/691	Serves the same purpose as Main-old, but uses chord space midground generation.
Main-phil.lhs	542, 690/691	Gives a sample performance of the composition from Main-old.lhs.
Midground.lhs	690/691	Implements the grammar application algorithm.
NoteRules.lhs	690/691	Provides harmony restrictions for foreground generation.
OpticFuns.lhs	690/691	Implementation of OPTIC relations and functions.
OpticFuns2.lhs	690/691	Equivalence class representation for OpticFuns.lhs
OpticToNotes.lhs	690/691	Same as ToNotes.lhs but for chord space midground generation.
PhilPerf.lhs	542	Defines the "Phil" algorithm used by main-phil.lhs.
Rules.lhs	690/691	Defines the grammar used by midground.lhs.
Structure.lhs	690/691	An unfinished implementation for background generation and let-statements.
ToNotes.lhs	690/691	Functions for converting midground output other data types.
ToNotes2.lhs	690/691	Functions for converting foreground output to Music1 data.
Utility.lhs	542/690/691	A collection of general functions used by other modules.

Four additional source code files are included that generate a longer composition of the form A-A'-B-A-C-D-A-A'-B-A, where the A-A'-B-A sections are the same as those defined in MainOld and MainOPTIC, and the sections C and D are 16 measures each. Section C is in the relative major and section D is in the subdominant.

Filename	Origin	Description
Exp.lhs	690/691	Definition of the C and D sections with the old midground implementation.
ExpOPTIC.lhs	690/691	Definition of the C and D sections with chord spaces.
ExpComp.lhs	690/691	Definition of the C and D sections with the old midground implementation.
ExpCompOPTIC.lhs	690/691	Definition of the C and D sections with chord spaces.

References

[1] Andres Garay Acevedo. Fugue composition wtih counterpoint melody generation using genetic algorithms. *Lecture Notes in Computer Science*, 3310, 2005.

[2] David D. Boyden. *A Manual of Counterpoint Based on Sixteenth-Century Practice*. Carl Fischer Music, New York, 1970.

[3] Clifton Callender, Ian quinn, and Dimitri Tymoczko. Generalized voice-leading spaces. *Science Magazine*, 320(5874):346–348, 2008.

[4] Bradley J. Clement. Learning harmonic progression using markov models. In *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1998.

[5] Richard Cohn. Neo-riemannian operations, parsimonious trichords, and their tonnetz representations. *Journal of Music Theory*, 41(1):1–66, 1997.

[6] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Prentice Hall, New Jersey, 1999.

[7] Kemal Ebcioglu. An expert system for schenkerian synthesis of chorales in the style of j.s. bach. In *ICMC 1984*, 1984.

[8] Jean-Francois Paiement et al. A probabilistic model for chord progressions. In *Proceedings of the 6th International Conference on Music Information Retrieval*, 2005.

[9] Jean-Francois Paiement et al. A distance model for rhythms. In *Proceedings of the 25th International Conference on Machine Learning*, 2008.

[10] Michael Towsey et al. Towards melodic extension using genetic algorithms. *Educational Technology and Society*, 4(2), 2001.

[11] Steven Brown et al. Music and language side by side in the brain: a pet study of the generation of melodies and sentences. In *European Journal of Neuroscience*, 2006.

[12] Michael Gogins. Score generation in voice-leading and chord spaces. In *ICMC 2006*, 2006.

[13] James Greeson. 18th century counterpoint (online book). `http://www.uark.edu/ua/muth/counterpoint/10_Fugues/10_Fugues.pdf`, 2006.

[14] Rachael Wells Hall. Geometrical music theory. *Science Magazine*, 320(5874):328–329, 2008.

[15] Rachel W. Hall. Playing musical tiles. In *Bridges: Mathematical Connections in Art,Music, and Science*, 2006.

[16] Phillip B. Kirlin and Paul E. Utgoff. A framework for automated schenkerian analysis. In *ISMIR 2008*, 2008.

[17] William Renwick. *Analyzing Fugue: a Schenkerian Approach.* Pendragon Press, Stuyvesant, NY, 1995.

[18] Stephen W. Smoliar. A computer aid for schenkerian analysis. In *Proceedings of the 1979 Annual ACM Conference*, 1979.

[19] Patrick Suppes, Lin Liang, and Michael Bottner. Machine learning comprehension gramamrs for ten languages. *Computational Linguistics*, 22, 1996.

[20] Dimitri Tymoczko. The geometry of musical chords. *Science Magazine*, 313(5783):72–74, 2006.

[21] Peter Worth and Susan Stepney. Growing music: musical interpretations of l-systems. *Applications on Evolutionary Computing*, 2005.

[22] Liangrong Yi and Judy Goldsmith. Automatic generation of four-part harmony. In *UAI Applications Workshop 2007*, 2007.

[23] Jason Yust. The geometry of melodic, harmonic, and metrical hierarchy. In *Mathematics and Computation in Music*, 2009.

[24] Marek Zabka. Generalized tonnetz and well-formed gts: a scale theory inspired by the neo-riemannians. In *Mathematics and Computation in Music*, 2009.