MUSASSIST: A DOMAIN SPECIFIC LANGUAGE FOR MUSIC NOTATION

Ilana Shapiro
Pomona College
Claremont, USA
`issa2018@mymail.pomona.edu`

ABSTRACT

MusAssist is an external, declarative, domain specific language for music notation that bridges the abstraction gap between Western music theory and composition. Users can describe unique high-level templates for chords and arpeggios (all triads and seventh chords), scales (diatonic, chromatic, and whole tone), the five primary cadences, and the four primary harmonic sequences with desired length. Distinctively, MusAssist matches the level of abstraction of a template to the theoretical musical structure it describes (e.g. users can specify a harmonic sequence without needing to manually expand it to the chords and notes it comprises). Thus, users can write out specifications precisely at the conceptual levels of the musical structures they would organically conceive when composing by hand. In MusAssist, users can also change key signatures, start a new measure, and describe fundamental musical objects such as notes, rests, and customized chords. The Haskell-based MusAssist compiler expands high-level templates (thus lowering the level of abstraction to individual notes) and translates the program to MusicXML, a language accepted by most major music notation software, for further manual editing and playback.

1. INTRODUCTION

When writing music in the framework of Western music theory, composers manually transition from theoretical musical concepts to notes on a page. This process can be slow and tedious, requiring the composer to expand complex musical structures by hand, such as cadences and harmonic sequences, to the individual notes they define. Therefore, the level of abstraction of the musical structure is higher than what the composer writes.

Domain specific languages (DSLs) are programming languages highly specialized for a specific application and thus characterized by limited expressiveness. An *external* DSL has custom syntax that is separated from the primary language of its application.

This paper presents MusAssist, an external, declarative DSL for music notation that bridges the abstraction divide between Western music theory and notation. Users describe a composition in MusAssist's straightforward, high-level syntax, modeled around the musical elements composers organically conceive when writing by hand, and the MusAssist compiler automates the expansion of these elements to their constituent notes. MusAssist's declarative programming paradigm was chosen to correspond with the lack of control structures in handwritten music.

MusAssist is unique in that users can encode specifications for complex musical templates at the same level of abstraction as the theoretical musical structures they describe. Specifically, users can specify high-level templates for chords and arpeggios (major, minor, dominant, augmented, half diminished, and diminished triads and seventh chords in any inversion), scales (major, natural/harmonic/melodic minor, chromatic, and whole tone), cadences (perfect authentic, imperfect authentic, plagal, half, deceptive), and harmonic sequences (ascending fifths, descending fifths, ascending 5-6, descending 5-6) of a desired length. MusAssist also supports individual notes, rests, and customized chords consisting of user-defined collections of notes and enables the user to change the key signature or start a new measure. All high-level templates are expanded, lowering the abstraction level to notes, by the Haskell-based MusAssist compiler.

The target language of the MusAssist compiler is MusicXML, itself a DSL that is an extension of XML (Extensible Markup Language). MusicXML is accepted by most major notation software, such as MuseScore. Thus, users can open the resulting MusicXML file of a compiled MusAssist composition in MuseScore or another program for further customization, editing, and playback. MusAssist thus serves as a professional music compositional aid, filling in an integral part in the composition process where Western composers must make the transition from the theoretical, high-level musical framework with which they describe their piece to the notes that these structures constitute. The subsequent ability of users to manually edit the compiled MusAssist composition speaks to MusAssist's present role as an assistant in the creative process, rather than a fully expressive means of music creation. MusAssist may also be particularly helpful to Western music theory students as an educational tool, enabling them to visualize the relationship between a Western theoretical musical structure and its expanded form, such as in conceptualizing the chords resulting from the expansion of a cadence.

This paper first summarizes related work in music notation DSLs. Then, features of the MusAssist language are outlined, followed by the presentation of a use case. Finally, the MusAssist compiler structure and the central logic behind its automated template expansions are described.

2. RELATED WORK

The era of music programming languages began in 1957 with Max Mathews' MUSIC-N languages at Bell Labs, developed on individual IBM mainframes. The first of these languages, Music I, generated a single, equilateral, triangular waveform with identical rise and decay characteristics. It was capable of playing only melody: the user specified pitch, amplitude, and duration for individual notes. Music II supported four independent voices of sound and a choice of 16 waveforms, and Music III introduced the concept of the *unit-generator*, the building block for sound synthesis programming languages that corresponds to the functions of analog synthesizers. Music IV was a more computationally sophisticated version of Music III, and machine independence was finally reached in 1966 with Music V, which became the first publicly distributed music programming language [1].

Since then, computer scientists have taken advantage of the increased flexibility afforded to DSLs via their limited expressiveness to create music DSLs tailored towards notation, algorithmic composition, sound synthesis, live coding with music performance, and more. Aside from MusAssist, MusicXML, LilyPond, and PyTabs are commonly used music DSLs also specializing in notation.

Michael Good's MusicXML is an Internet-friendly, XML-based, declarative DSL that represents standard Western music notation and scoring practices. Similar to how the popular MIDI format helped create a standardized format for electronic instruments, MusicXML provides a standardized format for online sheet music to create a consistent method for representing complex, structured musical data. MusicXML thus introduces smooth interchange between musical applications specializing in notation, performance, analysis, music information retrieval, and more [2].

To achieve this, Good derives MusicXML from XML due to its Internet-friendly nature, straightforward use in document creation, and human readability. In contrast, MIDI is very difficult to read and write, and is also less powerful and expressive than XML [3].

MusicXML is more expressive than MusAssist, but the abstraction level of all musical elements is extremely low (i.e. chords must be written out as individual notes) and though easily readable, its syntax is cumbersome and tedious to write by hand. However, its enhanced expressiveness makes MusicXML an excellent target compilation language for MusAssist's user-friendly syntax and high-level Western theoretical musical templates.

LilyPond, an external, declarative DSL created by Han-Wen Nienhuys and Jan Nieuwenhuizen, is similar to MusAssist. It features a "modular, extensible and programmable compiler" written in Scheme to generate Western music notation of excellent quality, and supports the mixing of text and music elements. Text-based musical expressions, or fragments of music with set durations, are compiled to an aesthetically formatted score [4].

LilyPond and MusAssist are both music notation DSLs tailored to non-programming audiences. However, they differ in two fundamental areas: (1) MusAssist supports complex music templates at the levels of abstraction of the musical structures they represent, whereas LilyPond only supports granular, low level composition of individual notes and chords, and (2) the output of the MusAssist compiler is intentionally editable via notation software, unlike LilyPond's compiler, which produces a static, printable PostScript or PDF file by taking in a file with a formal representation of the desired music [4].

Simic et al.'s external, declarative DSL PyTabs is also geared toward music notation, but in a different domain than MusAssist. Specifically, the authors attempt to solve the visual problem of tablature notation and the lack of standardization in specifying tablature-based note durations by consolidating these issues into a formal language. However, tablature notation is outside the scope of MusAssist's text-based Western theoretical musical structures [5].

3. LANGUAGE FEATURES

3.1 Low-Level Fundamentals

On the most basic level, MusAssist supports individual rests and notes. Rests are given a duration from sixteenth to whole note, and notes are further defined by note name (A to G), accidental (double flat to double sharp), and octave (1 to 8, after the range of a piano). Just as in traditional Western notation, the absence of an accidental indicates natural quality. Users can also define customized chords, or user-defined lists of individual notes. These are not considered templates as the high-level description of the chord is not given, and the granularity is at the note level.

3.2 High-Level Templates

MusAssist supports templates for chords, arpeggios, scales, cadences, and harmonic sequences, specified uniquely at the abstraction levels of the Western theoretical musical structures they represent.

Following the principles of Western music theory, chords are specified by their root note, quality (major, minor, augmented, dominant, diminished, or half diminished), inversion (root, first, second, or third), and chord type (triad or seventh). Half diminished, dominant, and third inversion options can only apply to seventh chords. The root note cannot have a double accidental, as this can introduce triple accidentals in the chord, which MusAssist does not support.

Arpeggios are defined with a similar specification to MusAssist chords, since according to Western music theory, an arpeggio is simply the notes of a chord played individually in sequence. However, unlike with chords, for arpeggios the user also supplies direction (ascending or descending).

Diatonic scales are given by scale type (major or natural/harmonic/melodic minor) and key, while non-diatonic scales are simply specified by their type (chromatic or whole tone). A scale is either ascending or descending, must be given a length, and does not necessarily begin on the tonic – a start note must be supplied. Following convention, chromatic scales are notated with sharps when ascending and flats when descending.

Cadences are specified by cadence type (perfect/imperfect authentic, half, plagal, or deceptive) and key.

Currently, MusAssist only supports a single treble clef line. Thus, cadences are written out in the upper voices only, in keyboard voice leading style, and incorporating principles of smooth voice leading.

Based on the principles of Western functional harmony, there are several ways to represent a cadence. In MusAssist, the representations in Table 1 were chosen. In each row of Table 1, the major version is presented first, with the minor version following in parentheses.

Perfect Authentic	IV-V-I (iv-V-i)
Imperfect Authentic	IV-vii$^{o\,6}_{\ \ 4}$-I$^6_4$ (iv-vii$^{o\,6}_{\ \ 4}$-i$^6_4$)
Plagal	IV$^6_4$-I (iv$^6_4$-I)
Deceptive	IV-V$^6_4$-vi$^6_4$ (iv-V$^6_4$-VI$^6_4$)
Half	IV-ii^6-V (iv-ii^{o6}-V)

Table 1: MusAssist Cadences Summary

All cadences except perfect authentic are built exclusively with triads. Although MusAssist does not currently support a bass line, in order to simulate the requisite 4-5-1 bass line for perfect authentic cadences, the root in the final chord is doubled. This also allows for the uppermost voice to follow the requisite 2-1 downward step in the final two chords of the cadence without compromising the root position of the final chord. The perfect authentic cadence is demonstrated in Figure 1, produced with the MusAssist syntax (`Perfect Authentic Cadence, Eb5 minor, sixteenth`) compiled and loaded into MuseScore notation software.
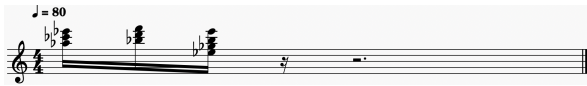
Figure 1: Perfect Authentic Cadence in E♭ minor

Finally, harmonic sequences are specified by harmonic sequence type (ascending fifths, descending fifths, ascending 5-6, or descending 5-6), key, duration of each chord, and length of the sequence. Since MusAssist does not yet support multi-line composition, harmonic sequences are written like cadences in keyboard-style voice leading.

In Western music theory, harmonic sequences can be implemented in several ways depending on the desired inversion scheme. Though the upper-voice harmonization of a harmonic sequence need not follow the direction in the sequence's name, MusAssist chooses a chord inversion and voice leading pattern such that each sequence does so. For instance, the Ascending Fifths sequence will ascend, and the Descending 5-6 sequence will descend. Each pattern also maximizes smooth voice leading.

The chosen patterns for each MusAssist sequence are summarized in Table 2. All sequences are shown in major in this demonstration, but their minor counterparts are also supported. Each sequence consists of fourteen distinct chords before repeating in the subsequent octave.

Ascending Fifths	I$^6_4$	V	ii$^6_4$	vi	iii$^6_4$
	vii^o	IV$^6_4$	I	V$^6_4$	ii
	vi$^6_4$	iii	vii$^{o\,6}_{\ \ 4}$	IV	
Descending Fifths	I	IV$^6_4$	vii^o	iii$^6_4$	vi
	ii$^6_4$	V	I$^6_4$	IV	vii$^{o\,6}_{\ \ 4}$
	iii	vi$^6_4$	ii	V$^6_4$	
Ascending 5-6	I	vi^6	ii	vii^{o6}	iii
	I^6	IV	ii^6	V	iii^6
	vi	IV^6	vii^o	V^6	
Descending 5-6	I$^6_4$	V	vi$^6_4$	iii	IV$^6_4$
	I	ii$^6_4$	vi	vii$^{o\,6}_{\ \ 4}$	IV
	V$^6_4$	ii	iii$^6_4$	vii^o	

Table 2: MusAssist Harmonic Sequences Summary

3.3 Additional Features

Beyond compositional elements, users can set the key signature at the start of any measure up to seven sharps or flats by specifying note name, accidental, and quality (sharp or flat). Users can also start a new measure or create a blank measure. Finally, users can assign MusAssist expressions to string labels and reuse them later in the program (the labels are syntactic sugar for the expressions). MusAssist comments are designated with the double slash //.

The tempo for all MusAssist programs is set to ♩ = 80 BPM and cannot currently be changed. This also applies to the time signature, which is set to $\frac{4}{4}$.

All compiled MusAssist programs adhere to standard notation conventions. Notes and rests are broken over barlines as well as over the strong beat (beat three) of the measure. They are divided greedily into valid rhythmic units (from sixteenth to whole note) ordered either least to greatest, or greatest to least in the case of spillage of a tied note over the barline into the following measure.

4. SAMPLE PROGRAM

The full breadth of MusAssist's syntax is demonstrated in Figure 2, and Figure 3 presents the resulting compiled MusicXML code when opened in MuseScore.

Several features of MusAssist are clarified in Figures 2 and 3:

- The key signature can be changed consecutively as many times as desired, but only the last will take effect (as seen m. 1 and m. 12 of Figure 3). Changing the key signature also triggers a new measure.

- Note durations are automatically broken by the compiler both on the strong beat and on the barline (such as in mm. 2-4 of Figure 3).

- Labeled phrases are not notated until the label is referenced, rather than defined.

- The difference between a customized chord and a chord template is exemplified on line 14 of Figure 2.

```
 1 SET_KEY A major
 2 SET_KEY A major
 3 SET_KEY G major
 4 (D4 whole) (F#4 quarter) (Ab4 dotted_quarter) (G#4 eighth) (rest sixteenth)
 5 // note without b or # is natural
 6 notes1 = (D4 whole) (F#4 quarter) (Ab4 quarter) (G#4 eighth) (rest whole)
 7 chords1 = ([Bbb5, Db5, C5] half) ([C#5, E5] half)
 8 chord = (D6 minor arpeggio, root inversion, eighth)
 9 (F#4 dominant seventh chord, third inversion, eighth)
10 (D4 whole) (F#4 dotted_quarter) (Ab4 quarter) (G##4 eighth) (rest sixteenth)
11 NEW_MEASURE
12 NEW_MEASURE
13 ([Bbb5, Db5, C5] half)
14 ([C#5, E5] half) (C6 minor triad, first inversion, dotted_eighth)
15 (F#4 half diminished seventh chord, second inversion, eighth)
16 (D#4 augmented seventh arpeggio, root inversion, quarter)
17 SET_KEY D minor
18 SET_KEY C# major
19 (C harmonic minor descending scale, startNote = Eb4, quarter, length=10)
20 (Descending Fifths Sequence, G5 minor, quarter, length=15)
21 (Perfect Authentic Cadence, D#5 major, half)
22 notes1 chords1 (Ascending Fifths Sequence, G#3 minor, quarter, length=5)
23 chord (Deceptive Cadence, Eb5 minor, sixteenth) chords1
```

Figure 2: MusAssist Syntax

Figure 3: Compiled MusAssist Program in MuseScore

5. COMPILER STRUCTURE

The MusAssist compiler is written in Haskell. Its high-level structure is as follows:

1. MusAssist's concrete syntax is parsed into abstract syntax, represented as Haskell algebraic data types (ADTs). Parser combinators were chosen for their flexibility and ease of customization. Parsec, an industrial strength parser library, is used, and Parsec's helper module Token handles lexing. The parse preserves the abstraction level of all templates.

2. All templates, now represented as ADTs, are expanded using the logic in Section 6 until the granularity reaches the note level. The result of this intermediate stage is abstract syntax whose abstraction level matches that of the target language, MusicXML.

3. The low-level abstract syntax resulting from the fully expanded templates is translated to MusicXML. This step contains the temporal logic that subdivides notes and rests across barlines and strong beats.

The resulting MusicXML file can then be opened in standard music notation software like MuseScore for viewing, further editing, and playback.

6. TEMPLATE EXPANSION LOGIC

MusAssist's distinguishing feature is its ability to automate the expansions of Western theoretical musical templates given user-supplied specifications at a higher level of abstraction. The logic underlying the expansions is summarized below.

6.1 Backbone Logic

6.1.1 Generating Notes in a Diatonic Scale

Most MusAssist templates are built upon the diatonic scale. To automate the expansions of these templates, we must first be able to generate a note in a desired diatonic scale given a positive interval within one octave of the specified tonic. Recall that a MusAssist note is defined by note name, octave, and accidental. Given the target interval n, to determine the note name we begin at the tonic and travel n steps up MusAssist's custom Haskell ADT for note names, a circular Enum instance ordered as the C major scale is.

The desired octave is either the same as that of the tonic, or one greater if the desired note name (disregarding accidental) comes before the tonic note name in the C major scale. For instance, as seen in Figure 4, the red note names D, E, and F come before G in the C major scale, and the octave number of each is one higher than the tonic in a G major scale.

Figure 4: Octave Analysis of G Major Scale

To determine the desired accidental, first realize that in any key, a perfect interval will generally have the same accidental as the tonic, with two exceptions: the perfect fifth above B is F♯, and the perfect fourth above F is B♭. To work out the logic behind the accidental of a desired imperfect interval, consider Figure 5. Here, we enumerate all single-accidental key signature names (even invalid ones that contain double sharps or flats) in order to establish the pattern. Key signature names are grouped under the accidental of the note that is the desired interval from the tonic. For instance, in the key of A♭, the major second interval from the tonic is B♭. The accidental of B♭ is ♭, so A♭ falls under the ♭ column in Figure 5.

Major Seconds			
♭	♮	♯	𝄪
F♭	E♭	E	E♯
C♭	B♭	B	B♯
G♭	F	F♯	
D♭	C	C♯	
A♭	G	G♯	
	D	D♯	
	A	A♯	

Figure 5: Accidental of Major Second from Tonic per Key

From Figure 5 we see that given any key, the major second above the tonic has the same accidental as the tonic, except for any key with E or B in its name. In these keys, the accidental is "lifted" (i.e. ♭→ ♮, ♮→ ♯, and ♯→ 𝄪).

A similar pattern emerges for minor thirds, major sixths, and minor sevenths from the tonic. Using the result of this analysis, we can determine the accidentals of the inverse qualities (i.e. major versus minor) of the imperfect intervals by either lowering the computed accidental when going from major to minor, or lifting it otherwise. Augmented and diminished intervals from the tonic are not considered since they do not appear in diatonic scales.

6.1.2 Generating Chord Templates in a Diatonic Scale

Generating chord templates in a diatonic scale becomes relevant for the expansions of the highest-level MusAssist templates – namely, cadences and harmonic sequences – that are defined in Western music theory by lists of chord templates rather than notes.

The goal here is to automate the generation of chord templates for triads in a diatonic scale given a specified tonic tone and quality for the scale (major or minor), inversion, and positive interval within one octave of the tonic for the chordal root. If we need to generate a chord template for a seventh chord, we simply generate the base triad and add the fourth note afterwards.

In order to complete the triad template definition from the supplied information, we simply need to determine the desired chord quality. Western music theory dictates that the major diatonic scale contains the triads I-ii-iii-IV-V-vi-vii°, and the minor diatonic scale contains the triads i-ii°-III-iv-v-VI-VII. Using this, we can compute the triad quality given the tonic quality and the supplied interval between tonic and desired chordal root.

6.2 Template Expansions

6.2.1 Scales

The expansion of all major and natural/harmonic/melodic minor scales is derived from the logic in Section 6.1.1. The scale is generated in relation to its tonic, rather than the specified starting note. The tonic octave is always set so that the tonic falls below the start note. This ensures that the initial interval between tonic and start note is positive, with the interval then increasing for ascending scales and decreasing for descending scales until the desired scale length is reached. If the tonic is reached in the scale generation, we reset the tonic to be one octave higher or lower (corresponding to scale direction) so that the interval between the next note in the scale and the current tonic is always positive and within a single octave. Finally, consider that all minor scales are treated as natural when generating their notes. If needed, the notes are modified afterwards in order to appropriately raise the sixth and/or seventh scale degree(s) for harmonic and melodic minor scales.

For the non-diatonic scales (chromatic and whole tone), the C below the start note is set as the "tonic" in order to determine the octave of each note in the scale generation. As with diatonic scales, the tonic octave is appropriately shifted one octave higher or lower, corresponding to scale direction, if it is reached during the scale generation.

As seen in Figure 6, for 10 of the 12 tones, chromatic scales "double" the note name, with the directional accidental (sharp for ascending, flat for descending) falling on the second occurrence. The two exceptions marked in red in Figure 6 are E and B in the ascending version, and C and F in the descending version.

Figure 6: Chromatic Scales

If we are on a single note, we simply move up the scale. Otherwise, we repeat it and insert an accidental on the second occurrence.

Whole tone scales are constructed with a similar model. As seen in Figure 7, the ascending whole tone scale is missing the note B, while the descending is missing C.
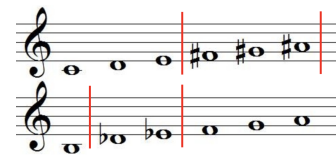

Figure 7: Whole Tone Scales

To determine the next note name, we simply traverse up or down the C major scale, excluding the appropriate tone. To determine the next accidental, we accordingly lift or lower the current accidental if we reach one of the red boundaries in Figure 7, otherwise leaving it unchanged.

6.2.2 Chords and Arpeggios

Each note in a chord or arpeggio is generated with the logic from Section 6.1.1 based on its interval from the tonic (i.e. the chordal root). Chordal thirds, fifths, and sevenths have respective intervals of 2, 4, and 6 from the tonic.

The imperfect chordal intervals are initially set to major for major, dominant, and augmented chords, and to minor for minor, half diminished, and fully diminished chords. Thus, after the note is generated, for augmented chords the chordal fifth accidental must be lifted, and the seventh must be lowered. For dominant seventh chords, the seventh must be lowered. Finally, for diminished chords, the fifth and seventh must be lowered, and for half diminished seventh chords, the fifth alone must be lowered.

To handle inversions, notice that the generated chord (whether triad or seventh) starts out in root position. Let n be the desired inversion value (0 for root, 1 for first inversion, 2 for second, and 3 or third). By incrementing the octaves of the first n tones of the chord in root position, we obtain the correct inversion. This process is demonstrated in Figure 8 with a C dominant seventh chord in third inversion.
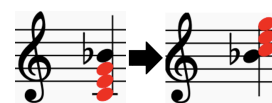

Figure 8: Chord Inversion Analysis

If we instead have an arpeggio, we first generate the notes in the underlying chord using the above logic. If the arpeggio is specified as a triad, we then double the first note of

the arpeggio (which is determined by the inversion) an octave higher or lower, according to the specified direction, in order to have a complete arpeggio. For instance, a C5 major descending arpeggio in first inversion would have E6 as its first note and E5 as its last note. Finally, we sort the tones in the arpeggio so that they strictly ascend or descend depending on the specified arpeggio direction, and return them as a list of individual tones, rather than a simultaneous cluster as in a chord.
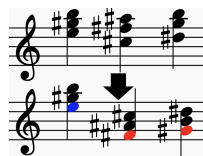
6.2.3 Cadences

A cadence must undergo two levels of expansion: an intermediate step from the cadence to the list of chord templates it defines, and a final step from chord templates to notes.

Recall that the chosen chords for each MusAssist cadence are defined in Table 1, which reveal the interval of each chordal root from the tonic. Using this, for each chord in the cadence, we first employ the logic from Section 6.1.1 to generate the root note of each chord. Then, we use the logic in Section 6.1.2 to generate a template for each chord, which then undergoes a second expansion in Section 6.2.2.

The scale quality supplied for the chord template generation in Section 6.1.2 is generally that of the cadence. However, there are exceptions. All V chord templates are generated within a major scale, no matter the cadence quality, since cadences always have major V chords. Similarly, we want the seventh triad in the imperfect authentic cadence to be diminished – i.e. built on the major seventh scale degree – no matter the local key quality, since we raise the leading tone in minor keys when moving towards the tonic. Thus, this chord template must also be generated within a major scale in Section 6.1.2.

The tonic octave supplied to Section 6.1.2 is also usually that of the cadence. However, in order for the cadences to follow smooth voice leading, the tonic octave must be lowered once in relation to the specified cadence octave when generating second inversion triads, which appear in all cadences except perfect authentic. This is demonstrated in the B major deceptive cadence in Figure 9. After converting to root position for clarity, note that the chordal root in the cadence octave is in blue and the roots an octave below (i.e. those of the second inversion triads) are in red.

**Figure 9**: Deceptive Cadence Octave Analysis

6.2.4 Harmonic Sequences

Like cadences, harmonic sequences are initially expanded to the chord templates they comprise, which then undergo a second expansion to notes. Recall from Figure 2 that each sequence consists of 14 chords, after which it cycles an octave above or below, depending on the direction of the sequence.

In order to generate chord templates for a sequence, we need to determine:

1. The interval between each pair of chordal roots, which dictates how to proceed from one chord to the next in the sequence

2. The inversion of each chord

3. The octave number of each chord (given by the chordal root octave) relative to the tonic

To determine (1) and (2), consider the analysis in Figure 10, which determines the interval pattern for the chordal roots of the descending 5-6 sequence (as defined previously in Figure 2), given the zero-based index of each chord in the sequence.
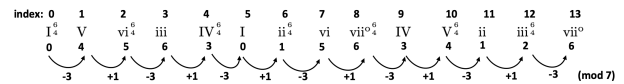
Figure 10: Descending 5-6 Interval Analysis

The top row of Figure 10 is the chord index, the second row is the chord progression from Table 2, the third row is the interval between each chordal root and the tonic, and the bottom row is the interval between each chordal root and the previous (modulo 7). A clear pattern for inversions (in the second row) and interval changes (in the fourth row) emerges based on the parity of the index. Identical analyses are applied to the remaining sequences to formalize their interval and inversion patterns.

Finally, we need to determine (3), the octave number of each chordal root relative to the tonic, or the root of the first chord in the sequence. Consider Figure 11, which presents an octave analysis for the descending 5-6 sequence.
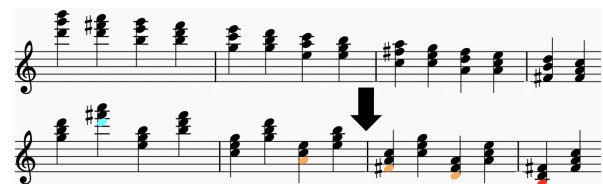
Figure 11: Descending 5-6 Octave Analysis

In Figure 11, all chords are converted to root position in order to visualize the octave numbers of their roots in relation to the tonic. The chordal roots in blue are an octave number above the tonic, the roots in yellow are an octave below, and the root in red is two octaves below. This same analysis is applied to determine the octave numbers of the chords in the remaining sequences.

Importantly, the interval analysis in Figure 10 holds for any representation of a harmonic sequence, as this is what defines the theoretical musical structure. However, the inversion and octave analyses in Figures 10 and 11 hold only for MusAssist's chosen representation of the sequences in Table 2, as a different inversion scheme would alter their outcomes.

7. CONCLUSION

This paper presents MusAssist, an external, declarative DSL for music notation that closes the abstraction gap between Western music theory and written composition. Users can uniquely write specifications in MusAssist's simple and

high-level syntax for scales, chords, arpeggios, cadences, and harmonic sequences at the precise levels of abstraction of the Western theoretical musical structures they describe. Fundamental musical elements such as notes, rests, custom note collections, new measures, and key signatures are also supported, along with the ability to reuse labeled expressions and indicate comments. MusAssist programs are translated by its Haskell-based compiler to MusicXML, enabling the composition to be loaded into notation software for further manual editing and playback. Thus, MusAssist serves the role of a tool for Western composers to use after they have mapped out the harmonic framework of their piece and must realize the structures they have described. MusAssist also has the potential to help Western music theory students easily generate expanded forms of musical structures as they learn, such as cadences, sequences, and scales.

Optimally, in the future MusAssist would support templates for additional non-diatonic structures like octatonic and pentatonic scales. Templates for key modulations are also planned, which would provide specifications that generate a sequence of chords modulating from a start key to a target key. Furthermore, future versions of MusAssist will allow for increased customizability of existing templates that are not currently fully expressive. The present MusAssist template expansions for cadences and harmonic sequences are limited to the representation schemes outlined in Tables 1 and 2. In order for MusAssist to fully close the abstraction gap between Western theoretical musical structures and their low-level notational forms, all functional harmonic variations should be supported. Additionally, support for two-clef, multi-staff composition would improve the implementation of cadences and harmonic sequences by including the essential baseline, and support for changing meter and tempo would provide users with increased compositional flexibility.

MusAssist would also benefit from veering beyond Western tonal theory and into other realms such as jazz by supporting all flavors of suspended, ninth, eleventh, and thirteen chords. Furthermore, MusAssist would ideally extend its support for non-diatonic structures beyond those encountered in Western music, such as the microtonal modal systems found in the Arabian-Persian $maq\bar{a}m$, and the intervallic patterns of the Indian $r\bar{a}ga$. Finally, additional user studies of MusAssist would give insight into potential improvements for language design.

Acknowledgments

8. REFERENCES

[1] C. Roads and M. Mathews, "Interview with max mathews," *Computer Music Journal*, vol. 4, no. 4, pp. 15–22, 1980. [Online]. Available: http://www.jstor.org/stable/3679463

[2] M. Good, "Musicxml: Introduction," Apr 2013. [Online]. Available: https://www.musicxml.com/publications/makemusic-recordare/notation-and-analysis/introduction/

[3] ——, *MusicXML: An Internet-Friendly Format for Sheet Music*, Dec 2001. [Online]. Available: https://michaelgood.info/publications/music/musicxml-an-internet-friendly-format-for-sheet-music/

[4] H.-W. Nienhuys and J. Nieuwenhuizen, *LilyPond, A System for Automated Music Engraving*, May 2003. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6160&rep=rep1&type=pdf

[5] M. Simić, Z. Bal, I. Dejanovic, and R. Vaderna, "Pytabs: A DSL for simplified music notation," *ResearchGate*, 2015. [Online]. Available: https://www.researchgate.net/publication/312607043_PyTabs_A_DSL_for_simplified_music_notation