Procedural Music Generation with Grammars

Lukas Eibensteiner, BSc*
Supervised by: Mag. Martin Ilčík†

Institute of Visual Computing and Human-Centered Technology
University of Technology
Vienna / Austria

Abstract

In this work we present an implementation of a generative grammar for procedural composition of music. Music is represented as a sequence of symbols with user-defined parameters, such as duration, pitch, harmony, volume, or active instruments. These sequences are developed by stochastic application of production rules that mutate, clone, or split their input. The generated sentences are converted to an output format (MIDI) using mapping functions. We demonstrate the system's capabilities through the generation of various types of musical structures, including phrase patterns, chords, melodies, and metrical hierarchies.

Keywords: music, procedural, formal grammars, production rules

1 Introduction

Procedural composition of music comprises various techniques that allow the generation of music from an abstract set of rules without direct involvement of a human composer. The efforts of the composer are shifted from manual placing of notes to the conception of the rules guiding the composition process. Besides being potentially cheaper to produce on a large scale, procedural music is especially interesting for interactive media such as video games, where the algorithm can adapt to dynamic events and user interaction. The idea of composing music through algorithms goes back centuries. In 1787 Wolfgang Amadeus Mozart published instructions for writing a waltz by rolling two dice [19]. The rolled numbers could be matched to entries in a table of musical bars that was provided with the instructions. By repeatedly throwing the dice and selecting the corresponding bars, one would end up with a new unique song.

A random shuffling of bars written by a master composer will undoubtedly sound aesthetically pleasing, but a random sequence of notes will hardly qualify as music. It is a well-known claim that music and mathematics are related [9, 23, 24], but formalizing the composition process remains difficult. Machine learning approaches with Markov chains [1] or neural networks [4, 6, 7, 13, 11, 15] mostly rely on an existing body of music for training. The resulting models can yield impressive results, but their output is difficult to adjust and can be lacking in regards to originality and high-level structure. Evolutionary algorithms [3, 10, 14] require either a fitness function, which is difficult to formalize, or expensive human evaluation [2]. Cellular automata [21] provide a source of simple rhythmic patterns, but the approach also offers limited control over musical structure and style.

In *A generative theory of tonal music* Lerdahl and Jackendoff provide evidence for the appropriateness of grammars for modelling music [17]. While they used grammars for musical analysis, grammars can also be used for generation. Since all rules are explicit, generative grammars offer detailed control and transparency over output. The effects of changing rules is immediate and requires no additional training. Further, grammars support structure at any level due to the hierarchical nature of the derivation and they can produce output of any length and complexity.

Recent work in this field shows how generative grammars can be used to generate certain aspects of musical structure individually. The grammar presented in this work is generalized for symbols with arbitrary parameters. This allows the generation of different structures within a single grammar, such as phrase structure, melodic patterns, and metrical hierarchies. Further, we show how stochastic derivation can be controlled to allow for repeating parts. For pitch selection we present a system of musical scales that enables us to define pitch relatively without requiring knowledge of the current harmony.

2 Related Work

Many of the technical terms used in this work come from the definition of formal languages by Noam Chomsky [5] and some familiarity with these concepts is necessary to understand the following sections. Our approach uses a context-free (Type-2) grammar, with some context-sensitive aspects (Type-1).

Some work has been done that explores the general applicability of grammars for describing musical patterns and composition [17, 22]. While these works are not

*l.eibensteiner@gmail.com
†ilcik@cg.tuwien.ac.at

primarily concerned with a concrete generative approach, they give significant justification for the use of grammars in composition and also offer useful input for designing production rules.

Holtzman [12] uses a custom grammar definition language for composition that supports all levels of the Chomsky hierarchy. They present the generation of both micro- and macro-structure using abstract tokens. McCormack [18] uses L-Systems for generating sequences of pitch letters. Rhythm, harmony, or dynamics are not supported and the generated music is rather simplistic. Quick et al. [20] present a grammar system for generating harmonic progressions. The symbols consist of harmonies with associated durations. The encoding of the duration and harmony as individual symbol parameters is very similar to our approach.

3 Methodology

In this section we present the particularities of our approach. We represent music as a sequence of symbols. Each symbol holds information related to the musical domain, such as length, pitch, tempo, volume, or active instruments. When generating the sequence we repeatedly replace symbols with new symbols. An example of such a replacement is illustrated in Figure 1.

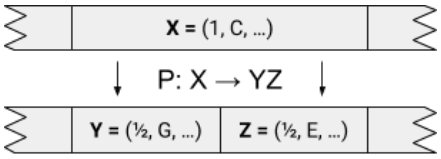

Figure 1: Production rule P that replaces a symbol X of length 1 and pitch C with two symbols YZ of length $\frac{1}{2}$ and a G pitch and E pitch respectively. In this case X is purely symbolic, a placeholder for YZ, and will not occur in the output sequence.

The symbols X, Y, and Z in Figure 1 come from a set that is called the *alphabet*. The fact that X is replaced with YZ is encoded as a *production rule*. The process of generating a sequence of symbols through the application of production rules is called *derivation*. It starts with a single dedicated symbol, the *axiom*, and continues until the sequence contains only *terminal* symbols. The terminal symbols are a subset of the alphabet and in our grammar exactly those symbols not accepted by any production rule. Our grammar generally corresponds to a *context-free* (Type-2) grammar [5].

3.1 Alphabet

For a grammar operating in the musical domain an alphabet Σ must be used that can be interpreted as music in some way, for example as notes, chords, or percussion beats. A simple option is to represent pitches using letter notation [18], so for melodies using the C major scale we could define it as $\Sigma := \{C,D,E,F,G,A,B\}$. A grammar with this alphabet would generate strings consisting of these seven pitches. Usually, a melody is not just a sequence of pitches; it has a rhythmical component as well, with notes of various lengths. Instead, we define the alphabet as the Cartesian product of multiple sets, for example $\Sigma = Pitch \times Length \times Volume$. The sets that make up this product can be freely defined, e.g. *Pitch* as pitch letters, *Length* as fractions, and *Volume* as an interval. Under this definition the symbols are n-tuples. Their entries will be referred to as *attributes* from here on.

3.2 Matching

In the formal definition of a context-free grammar, a production rule accepts a specific symbol and replaces it with a specific output sequence. Since the alphabet is defined as the Cartesian product of multiple, possibly non-discrete sets, defining production rules in this way is inefficient. Instead of a single symbol, each rule can match a subset of symbols, which we define using a Boolean-valued function on the alphabet, called a *predicate*. A rule accepts a symbol, iff its predicate returns *true* for that symbol. As a consequence, we define the output sequence as a function of the input symbol.

The predicate can be constructed using any of the symbol's attributes, which means the matching criteria can become arbitrarily complex. In practice, it is useful that one can quickly infer possible derivation paths from the grammar definition. Simple matching criteria should be preferred where possible. A useful method is having a string-valued *name* attribute and matching directly on its value, for example *all symbols named 'measure'*.

3.3 Randomness

The grammar is stochastic, which means there can be multiple rules that match a certain non-terminal. In such a case the derivation path is chosen randomly, weighted by a relative ratio associated with each rule. This allows us to vary the number of generated measures, the path of melodies, or rhythmic patterns in a controlled manner.

There is a tendency in music to repeat parts, phrases, and motives. A consequence of stochastic rule selection is that symbols that should represent repeating parts will be randomly matched by different rules during derivation, breaking their intended similarity. To solve this problem we define a numeric *seed* attribute for each symbol. Every time a rule needs to be picked randomly, the derivation algorithm initializes a random number generator using the seed of the current symbol. By using the same seed for two or more symbols we can guarantee a common derivation path, even if it contains random branches.

3.4 Commands

As mentioned above, a production rule is expressed as a function that maps a subset of the alphabet to a sequence of output symbols. These functions are defined through *commands*. The following list contains examples of basic command types:

mutator changes an individual attribute of the input symbol. Special cases of this type are *resize* for changing the length, or *rename* for changing the name.

repeat generates a specified number of copies of the input symbol.

split interprets a symbol as a 1D segment and divides it at specified offsets into two or more symbols. It must be specified how attributes are computed for the output symbols. For example, *length* can be shared proportionally and *volume* or *pitch* can be interpolated.

Additionally, there are meta commands that control the application of other commands. They can be used to express fixed derivation paths without the need for additional production rules:

chain takes a sequence of commands. The first command is applied to the input. All following commands are applied to the outputs of the previous one. This command allows the application of multiple commands within a single rule.

if applies a command to a symbol only if it matches an additional predicate.

branch applies an initial command and feeds each resulting symbol into a different sub-command. For example, one can *split* a symbol, and then apply different commands to the first and second output symbol.

3.5 Context-Sensitivity

Music seems to be context-sensitive to a certain degree. For example, in harmonic progressions the occurrence of a harmony is sometimes implied by previous harmonies [16]. For that reason we allow rules to access the previous and next neighbor of a symbol, both during matching and inside commands. More often than not, it is necessary to access the *terminal* neighbors of a symbol. We achieve this by allowing recursive derivation of symbols from within production rules. If a rule needs to know the terminal neighbor of a symbol it must first initiate and wait for the derivation of that neighbor. This can lead to deadlocks when a rule tries to recursively derive a symbol that is already waiting for the current symbol to be derived, but such circular dependencies can be detected automatically.

4 Implementation

Based on the theory in the previous section we developed a console application that compiles and executes musical grammar scripts. Each such script must declare the alphabet, initialize an axiom, define production rules, and specify a mapping from terminals to MIDI events. Both the application and scripts are written in C#, version 6.0 and 4.0 respectively.

4.1 Symbol Type

The system provides a basic symbol class with the most essential accessors, including the `Seed` for stochastic derivation, and `Name` for matching. Users can extend the basic symbol type and add custom attributes.

Generally, the user can declare attributes of any type. Most of the time using primitive types is sufficient, e.g. `float`, `int`, `bool`, `string`. The system also provides a handful of types specific to the musical domain, such as the `Pitch` type that can be constructed from scientific pitch notation (e.g. C4, G5, F#5) and allows easy conversion to MIDI numbers. Another example is the `Gesture` enumeration, which is used to specify whether a key was struck, held, or released. In Section 3 it was mentioned that some attributes should behave in certain ways during splitting. The system provides the following behaviors:

`Seed` provides a random number generator based on its current value. During a repeat or split command the new symbols receive a seed that is generated by the current seeds random number generator.

`Quantity` gets shared between the results of a split. An important use-case is the length of a symbol.

`Point` represents a value that exists only at a single point on the timeline. This type can be used to mark the position of beats, analogous to the click of a metronome.

`Interpolation` represents attributes that should be interpolated during a split. Possible applications include volume or pitch.

`Path` interpolates between two values by traversing a path between them in a weighted graph. This can be used for constructing harmonic progressions.

4.2 Domain Specific Language

Since C# does not have global functions, the bulk of scripting related functionality is contained in a single class. An instance of this class is accessible through the `Do` getter within the main method of each script. Wherever `Do.*` appears in any of the examples, it is a call on that instance. This removes the need for specifying the custom symbol type as a generic parameter for every single command, reducing visual noise. Users can easily add their own convenience methods to the `Do` object via extension methods.

4.3 Initializing the Axiom

The axiom is the starting symbol. When the grammar script is invoked, the program automatically creates an axiom by instantiating the user-defined symbol type with default values for all attributes. To initialize the axiom the user must call the `Axiom` method. Listing 1 shows a simple example of the axiom initialization.

```
Axiom(
    Do.Rename("song"),
    Do.Resize("8")
);
```

Listing 1: Code defining a chain of commands, which will be applied to the axiom before the grammar is invoked. In this case the name and length are initialized.

4.4 Defining the Grammar

The grammar is defined by passing a `Grammar` instance created by `Do.Grammar` to the script's `Apply` method. Listing 2 shows how a simple non-deterministic grammar with the axiom A and three production rules P_0, P_1, P_2 can be constructed.

```
Apply(Do.Grammar(
    // P0: A -> t (weight = 1.0)
    Do.Rule("A", 1.0,
        Do.Rename("t")
    ),
    // P1: A -> BB (weight = 3.0)
    Do.Rule("A", 3.0,
        Do.Rename("B"),
        Do.Repeat(2)
    ),
    // P2: B -> t
    Do.Rule("B",
        Do.Rename("t")
    )
));
```

Listing 2: Definition of a grammar that consists of three production rules.

`Do.Rule` creates a new production rule that uses the first string argument to match to the name of a symbol. The numeric weight is optional and set to 1.0 if unspecified. As a final input the method accepts a variable length list of individual commands. In this example only rule P_1 applies more than one command, since besides being renamed the symbol is also replaced with two clones of itself. In practice we would usually perform many more commands inside a single rule.

4.5 Relative Pitch

The system also provides a model for musical scales for working with relative pitches and intervals. A scale is represented by a function that maps an index $i \in \mathbb{Z}$ to a frequency. The frequency that is returned for index 0 is called the *tonic*. For example, in a chromatic scale (black and white keys on a piano) there are twelve semitones per octave. Assuming the tonic is A4 (440 Hz) it would map 0 to A4, 1 to A#4, 2 to B4, and so forth.

Scales can be built on top of each other. For example, the C-major scale uses only the white keys of the piano. It can be built on top of the chromatic scale, by skipping all indices that correspond to the black keys. The chromatic scale itself can be built upon the *cent* scale, with cents being a logarithmic unit of frequency (100 cent = 1 semitone) [8]. Figure 2 shows how a major scale with a tonic pitch of C4 is constructed.

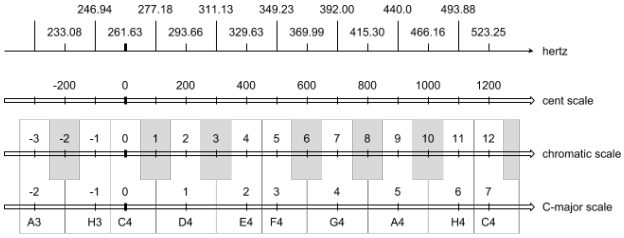

Figure 2: A C-major scale is created by only using pitches corresponding to white piano keys in the chromatic scale, which in turn only uses every hundredth pitch in the cent scale. The cent scale represents the top-level and its indices are directly converted to hertz. The tonic is defined as 261.63 Hz (C4).

Once the scale has been created, we can use integers relative to the tonic to produce melodies and chords. By adding an integer offset to all indices of a scale we can move the tonic. For example, adding an offset of 4 to the C-major scale would center the scale on its fifth pitch. It will still be comprised of the same pitches as before, but the index 0 now yields G4 rather than C4 and the mode will change from Ionian to Lydian. Doing that allows us to define production rules which are independent of the underlying harmonic progression. Moving the tonic enables us to achieve arbitrary levels of relativity when defining key, harmonic progressions, chords, and melodies.

4.6 Mapping Terminals

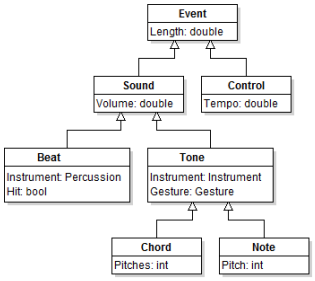

Figure 3: Simple class diagram of the four event types and their common base types. `Note` and `Chord` are used for monophonic and polyphonic instrument sounds. `Beat` represents a hit on a percussion instrument. `Control` is used for controlling meta parameters, such as overall tempo.

The last missing piece is how music is generated from the terminals. Since the terminals are all instances of a custom symbol type, the user must specify how they are mapped to a musical event. There are four types of such events which are shown in Figure 3. Once the derivation is complete, the mappings are applied to each symbol in the terminal sequence. In a final step the generated event sequences are written to individual tracks of a MIDI file.

Listing 3 shows how a symbol can be mapped to a monophonic piano melody and a bass drum beat.

```
// Piano melody
Map(x => new Note()
{
    Length = x.Length,
    Gesture = Gesture.Strike,
    Pitch = x.MelodyPitch,
    Instrument = Instrument.AcousticGrandPiano,
    Volume = 0.5
});

// Bass drum beat
Map(x => new Beat()
{
    Length = x.Length,
    Hit = x.BassDrumHit,
    Instrument = Percussion.BassDrum1,
    Volume = 0.5
});
```

Listing 3: Code that specifies two mapping functions, which map a terminal x to a piano note (first statement) and a bass drum hit (second statement). Users can choose which event properties are based on symbol attributes (e.g. Length) and which are hardcoded (e.g. Volume).

5 Results

We demonstrate the system's capabilities by means of multiple application examples. Each example in this section is focused on a specific musical idea or aspect, but they can be combined for more interesting results. For the sake of reproducibility all examples are deterministic. In practice many parameters can and should be randomized to achieve varying output.

5.1 Grouping Structure

In this example we generate a possible overall structure for a piece, what Lerdahl refers to as *grouping structure* [17]. It is a hierarchical division into groups of events that are perceived as belonging together. The example is based on a *Minuet in G Major, BWV Anh. 114* by Johann Sebastian Bach. By analyzing the piano score we can deduce a structure such as the one shown in Figure 4. The grammar for this example generates a sequence of eight phrases, each with a length of 8, and seed values that represent our structure. Based on this structure we could further define rules that split the phrases into measures, assign harmonies, and build a melody. By basing any randomization on the generated seeds, all phrases with the same seed will end up the same. The result is visualized in table 1. The seed values show the intended pattern.

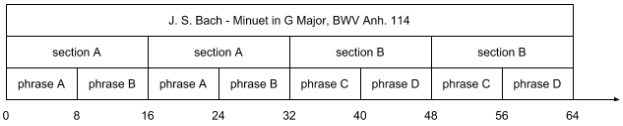

Figure 4: High-level grouping structure of a *Minuet in G Major, BWV Anh. 114* by Johann Sebastian Bach. The horizontal axis shows the offset in measures.

Name	Length	Seed
phrase	8	212140088
phrase	8	715172577
phrase	8	212140088
phrase	8	715172577
phrase	8	266780409
phrase	8	335640345
phrase	8	266780409
phrase	8	335640345

Table 1: This table represents the symbols in the output sequence. The seeds show the intended phrase pattern *ABABCDCD*.

5.2 Melody with Pitch Interpolation

In this example we generate a melody similar to that of a *Minuet in G Major, BWV Anh. 114* by Johann Sebastian Bach. The section of interest is shown in Figure 5.

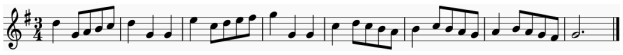

Figure 5: Right hand melody of measures 8 to 15 from Bach's *Minuet in G Major, BWV Anh. 114*.

We first need to analyze the melody to find possible generalizations. Measures 1, 3, 5, 6, and 7 all contain a quarter note followed by four eighth notes, a pattern which we name the *primary motive*. The four eighth notes will be referred to as *eighth-group* from here on. Listing 4 shows how we can split a measure into the quarter and the eighth-group and is visualized in *Step 2* of Figure 6.

```
Do.Rule("primary-motive",
    Do.Resize(3.0 / 4),
    Do.Branch(
        Do.SplitLength(1.0 / 4),
        Do.Rename("note"),
        Do.Rename("eighth-group")
    )
)
```

Listing 4: Rule generating the base pattern of our primary motive, which is a three-quarter measure that starts with a quarter note. It produces a *note* and an *eighth-group* symbol, the latter of which will be split further.

After the first quarter is split off, we need to apply a rule that generates our eighth-group. With attribute interpolation we only need to set the start and end pitch of the

symbol. The pitches inbetween will be interpolated when the symbol is split. This is shown in *Step 3* and *Step 4* of Figure 6. We still need to decide how the start and end pitches are determined.

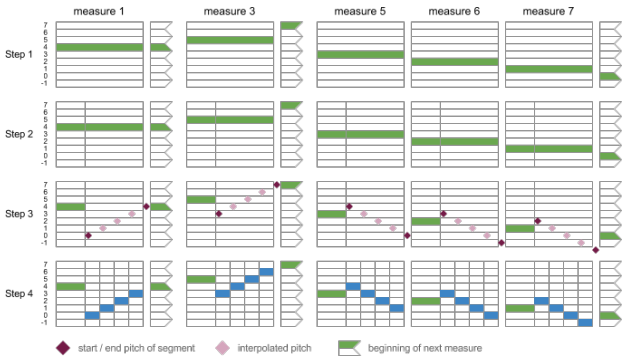

Figure 6: Generation of the primary motive.

```
Do.Rule("eighth-sequence",
    Do.Rename("note"),
    // Use pitch of next symbol if no end is specified.
    Do.If(x => !x.Pitch.UseEnd,
        Do.Apply(x => x.Pitch.UseEnd = true),
        Do.Apply(x => x.Pitch.End = x.Pitch.Value),
        Do.If(x => x.Next != null,
            Do.Apply(x => x.Pitch.End =
                x.Next.Pitch.Value)
        )
    ),
    Do.If(x => x.Pitch.Descending,
        Do.If(x => x.Smooth &&
              Math.Abs(x.Pitch.Gradient) < 4,
            // End group on one tone below next note.
            Do.Apply(x => x.Pitch.End -= 2)
        ),
        // Set start to be four notes above end.
        Do.Apply(x => x.Pitch.Value = x.Pitch.End + 4)
    ),
    Do.If(x => !x.Pitch.Descending,
        Do.If(x => x.Smooth &&
              Math.Abs(x.Pitch.Gradient) < 4,
            // End group on one tone above next note.
            Do.Apply(x => x.Pitch.End += 2)
        ),
        // Set start to be four pitches below end.
        Do.Apply(x => x.Pitch.Value = x.Pitch.End - 4)
    ),
    // Split into four notes.
    Do.Apply(x => x.Pitch.Interpolator = new Linear()),
    Do.SplitEven(4)
)
```

Listing 5: Rule generating a sequence of four notes of equal duration. Their pitches are each one step apart and lead right into the pitch of the next note.

The eighth-group is sometimes ascending, sometimes descending. We could deduce that if the first note of the measure is lower than or equal to that of the next measure, the sequence ascends (measures 1 and 3), otherwise it descends (measures 5, 6, and 7). Depending on that relationship we can start the sequence four pitches above or below the end pitch.

Further we can see that the eighth-group always leads into the following note. For the implementation of this rule we need to access the successor of the eighth sequence to see where it has to end. In measures 1 and 3 the first note before the eighth-group is not smoothly connected, but is followed by a jump in pitch. We use the custom boolean `Smooth` attribute as a flag to differentiate between the two cases. The implementation of all this can be found in Listing 5.

It is interesting that the rule in Listing 5 not only replicates, but also generalizes the primary motive. Depending on the initial and the following pitch our rule will try to find a sequence of four ascending or descending notes that connects the two pitches as smoothly as possible. Since all offsets are relative, the rule works for any pitch.

5.3 Chords and Arpeggios

In this example we generate chords and arpeggios from a harmonic progression and show how to use scales. First, the axiom is split into a chord and arpeggio half. Then, each half is split into four measures with individual harmonies, as seen in Figure 7. The harmonies are stored in the `Harmony` attribute as instances of our musical scale model. Listing 6 shows how chords can be generated.

Song							
Chords				Arpeggios			
I (0)	V (4)	VI (5)	IV (3)	I (0)	V (4)	VI (5)	IV (3)

Figure 7: Structure of the song generated in this example. The bottom row shows the individual measures and their harmonies.

```
// Use tonic (0), third (2), and fifth (4) for chord
Do.Apply(x => x.Pitches = x.Harmony.Pitches(0, 2, 4))
```

Listing 6: Command that generates a triad in a diatonic scale.

The arpeggios are generated in listing 7. Since arpeggios consists of multiple notes, the measure needs to be split further. The pattern is generated by branching on the results of the split and assigning a different pitch index to each part. Here the pattern is fixed, but it can be randomized by selecting from a set of pitches or using pitch interpolation. The result is shown in Figure 8.

```
// Two repetitions of the pattern per measure
Do.SplitEven(2),
// Pattern for arpeggio
Do.Branch(
    Do.SplitEven(4), // four notes
    Do.Apply(x => x.ArpeggioIndex = 0), // first note
    Do.Apply(x => x.ArpeggioIndex = 2), // second note
    Do.Apply(x => x.ArpeggioIndex = 4), // third note
    Do.Apply(x => x.ArpeggioIndex = 2)  // fourth note
),
Do.Apply(x => x.Pitches =
    x.Harmony.Pitches(x.ArpeggioIndex))
```

Listing 7: Sequence of commands that generates an arpeggio of a diatonic triad.

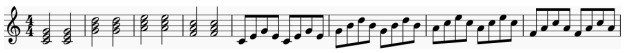

Figure 8: Generated chords and arpeggios.

5.4 Drum Set

In this example we generate a beat pattern with multiple instruments controlled by a single `Density` attribute. A low density should result in a slow pattern with fewer events, while a high value should generate more events and a pattern that feels more intense. Lerdahl found that metrical patterns contain strong and weak beats depending on their position in a hierarchy [17]. The first beat in a measure usually has the strongest accent. The next levels can be constructed by recursively splitting the measure into two or three equal parts. The positions of the splits mark the beats of each level. Listing 8 shows a recursive rule for generating these levels and storing them in the `BeatLevel` attribute.

```
Do.Rule("measure",
    Do.Apply(x => x.BeatLevel += 1),
    Do.Apply(x => x.BeatLevelCounter -= 1),

    // Since BeatLevel is a Point<int> the second
        split result
    // has a level of zero.
    Do.SplitEven(2),

    // Final depth of hierarchy is reached.
    Do.If(x => x.BeatLevelCounter <= 0,
        Do.Rename("beat"),
        // Make sure the level is at least 1 for every
            symbol.
        Do.Apply(x => x.BeatLevel += 1)
    )
)
```

Listing 8: Rule that generates a metrical hierarchy for a measure. The *BeatLevel* attribute stores the symbols level, with strong beats having the highest index.

We can compute a per-beat importance as a mixture of the normalized beat level and a global density value for the whole measure. By matching on different ranges of this importance value we can assign different instruments to different beats, for example a bass drum to the most important beats, and a hi-hat to beats with lower importance. The result is visualized in Table 2. The first measure only has beats on every quarter note, while the last measure shows an extremely dense beat pattern, so it seems to work just as intended. By randomly offsetting the beat importance by small values one can create a great variety of different patterns. This concept can be applied to all rhythmic aspects of music, such as harmony changes or notes in melodies.

5.5 Listening Samples

The previous examples in this section have a very narrow scope and are not interesting from a listener's perspective. The following files were all generated from a single grammar with five percussion mappings, one instrument mapping for chords, and one instrument mapping for the melody. The overall structure is *ABAB* with varying density values for each phrase as explained in Section 5.4. The chord and melody instruments are assigned randomly from a selection of options.

- Sample 1
  MIDI: `https://zenodo.org/record/1213678/files/620179078.mid`
  Sheet music: `https://zenodo.org/record/1213678/files/620179078.pdf`
- Sample 2
  MIDI: `https://zenodo.org/record/1213678/files/711202203.mid`
  Sheet music: `https://zenodo.org/record/1213678/files/711202203.pdf`
- Sample 3
  MIDI: `https://zenodo.org/record/1213678/files/711256687.mid`
  Sheet music: `https://zenodo.org/record/1213678/files/711256687.pdf`

6 Conclusion

We presented a generative grammar that can produce a wide range of musical structures. The use of custom symbols and mapping functions enables users to implement diverse musical ideas. The quality of the music depends on the quality of the rules, so one needs a firm grasp on music theory to define these grammars. Extending the built-in musical abstractions might open up the system to users with limited musical knowledge.

In the future we want to improve the symbol definition. The various different behaviors that attributes should expose during splitting could indicate some deeper issues with the current method. We also want to explore techniques to better encapsulate symbol attributes. Currently, attributes are declared globally, even though most rules only depend on a specific subset. Finally, defining harmonies and chord progressions using our scale model is often too rigid. Lerdahl provides a more flexible model in the form of pitch hierarchies [16], which could be incorporated in a future version.

References

[1] Charles Ames. The markov process as a compositional model: a survey and tutorial. *Leonardo*, pages 175–187, 1989.

[2] John Biles. Genjam: A genetic algorithm for generating jazz solos. In *Proceedings of the International Computer Music Conference*, pages 131–137. International Computer Music Association, 1994.

	Measure 1	Measure 2	Measure 3	Measure 4
Density	0.4	0.6	0.8	1.0
Level	5121312141213121	5121312141213121	5121312141213121	5121312141213121
Bass Drum	.....	x.....	x.....x.....	x...x...x...x...
Snare	x.....x.....	....x...x...x...	..x.x.x...x.x.x.	.xxx.xxx.xxx.xxx
Hi-Hat	x...x...x...x...	..x.x.x.x.x.x.x.	.xxxxxxx.xxxxxxx	.xxx.xxx.xxx.xxx
Crash	x.....	.....x.....	....x.....x...	..x...x...x...x.
Tom	....x...x...x...	..x.x.x...x.x.x.	.xxx.xxx.xxx.xxx	.x.x.x.x.x.x.x.x

Table 2: The *Density* and the 1-digit *Level* are combined to compute overall beat importance. The strings inside the cells of the bottom rows visualize the generated beat patterns for each instrument. `x` indicates that the instrument is struck, while `.` indicates that the instrument remains silent during that beat.

[3] Anthony R Burton and Tanya Vladimirova. Generation of musical sequences with genetic techniques. *Computer Music Journal*, 23(4):59–73, 1999.

[4] Chun-Chi J Chen and Risto Miikkulainen. Creating melodies with evolving recurrent neural networks. In *Proceedings of the 2001 International Joint Conference on Neural Networks (IJCNN-2001)*, pages 2241–2246. IEEE, 2001.

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

[6] Hang Chu, Raquel Urtasun, and Sanja Fidler. Song from pi: A musically plausible network for pop music generation. *arXiv preprint arXiv:1611.03477*, 2016.

[7] Douglas Eck and Juergen Schmidhuber. A first look at music composition using lstm recurrent neural networks. *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, 103, 2002.

[8] Alexander J Ellis. *On the musical scales of various nations*. Journal of the Society of arts, 1885.

[9] Thomas M Fiore. Music and mathematics, 2007.

[10] Andrew Gartland-Jones and Peter Copley. The suitability of genetic algorithms for musical composition. *Contemporary Music Review*, 22(3):43–55, 2003.

[11] Gaëtan Hadjeres and François Pachet. Deepbach: a steerable model for bach chorales generation. *arXiv preprint arXiv:1612.01010*, 2016.

[12] SR Holtzman. Using generative grammars for music composition. *Computer Music Journal*, 5(1):51–64, 1981.

[13] Allen Huang and Raymond Wu. Deep learning for music. *arXiv preprint arXiv:1606.04930*, 2016.

[14] Bruce Jacob. Composing with genetic algorithms. In *Proc. International Computer Music Conference (ICMC '95)*, pages 452–455. International Computer Music Association, September 1995.

[15] Vasanth Kalingeri and Srikanth Grandhe. Music generation with deep learning. *arXiv preprint arXiv:1612.04928*, 2016.

[16] Fred Lerdahl. *Tonal pitch space*. Oxford University Press, 2004.

[17] Fred Lerdahl and Ray Jackendoff. A generative theory of tonal music. 1983.

[18] Jon McCormack. Grammar based music composition. *Complex systems*, 96:321–336, 1996.

[19] Wolfgang A Mozart. Musikalisches würfelspiel: Anleitung so viel walzer oder schleifer mit zwei würfeln zu componieren ohne musikalisch zu seyn noch von der composition etwas zu verstehen. *Köchel Catalog of Mozarts Work KV1 Appendix 294d or KV6 516f*, 1787.

[20] Donya Quick and Paul Hudak. Grammar-based automated music composition in haskell. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 59–70. ACM, 2013.

[21] Paul Reiners. Cellular automata and music: Using the java language for algorithmic music composition. `https://www.ibm.com/developerworks/java/library/j-camusic/` (accessed October 19, 2016), 2004.

[22] Curtis Roads and Paul Wieneke. Grammars as representations for music. *Computer Music Journal*, 3(1):48–55, 1979.

[23] Kathryn Vaughn. Music and mathematics: Modest support for the oft-claimed relationship. *Journal of Aesthetic Education*, 34(3/4):149–166, 2000.

[24] Guy Warrack. Music and mathematics. *Music & Letters*, 26(1):21–27, 1945.