Temporal-Scope Grammars: A Musical Fragment Generator

Lukas Eibensteiner l.eibensteiner@gmail.com

May 16, 2021

Contents

1 Introduction

Temporal-scope grammars are a novel approach for automating the composition of polyphonic music. In this article, we will demonstrate the generation of motifs, high-level piece layouts, and chord progressions, culminating in the definition of a complete, polyphonic composition based on this approach. The resulting pieces will consist of multiple voices, each with a unique layout that divides the duration of the piece into a sequence of measures. In each measure we query a shared chord progression that we generate beforehand, which synchronizes the voices to a common harmonic context. The application of our polyphonic model shows that we can decouple voices and abstract musical aspects from each other, while not compromising on their synchronization and the quality of the compositions.

2 Construction

Our way of constructing the example in this section—starting at lower-level grammars and progressively integrating them into higher-level ones—goes in the opposite direction of the derivation process. The reason is that the lower-level grammars are more closely related to the generated music and thus easier to understand. The separate definition of these sub-grammars should further demonstrate that they are easy to reuse, recombine, and replace when we are not satisfied with the results. Transparency and loose coupling are, after all, primary advantages of grammars over other methodologies. A compact version of the code in this chapter, as well as a translation to TypeScript, can be found in Appendix A.

2.1 Motifs

We start by defining sub-grammars that generate motifs, which is the first level of abstraction above the note level. A motif is a short arrangement of sounds, no longer than a measure, which fits into the metric and harmonic framework, but usually has no further internal organization. Within a motif we may access metric and harmonic attributes such as *key*, *mode*, *harmony*, *tempo*, *beats*, and *beatType*, but we will avoid changing them, as the motif should not deviate from its context. Attributes that we will change are *time*, *span*, *gain*, and *freq*, as long as these changes take the context into consideration. For example, we can use metric units for measures, beats, and note fractions, but should avoid absolute and relative splits, as they are not aligned with the metric structure. We will define four different motif generators: BASS, PAD, LEAD, and DRUM.

2.1.1 Example: Helpers

Before we implement the actual motif generators, we define several helper functions that simplify their definition. CUT splits the entity into two parts (A, B) or (B, A), where A has a fixed size s and B fills the remaining space. Besides

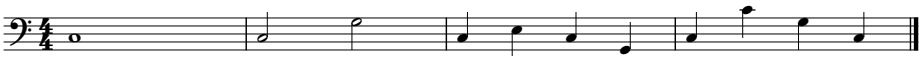

Figure 1: Each measure shows a possible result of the BASS grammar (File: bass.mp3). We normalized the results to C-major and common time.

their temporal scope, both parts are equivalent. We additionally trim the result, in order to guarantee that it fits into the input entity. REP repeats a part of a fixed size s until the input entity is covered and assigns the index to the *index* attribute. Again, we trim the result. DEG1 and DEG2 assign a random value to the *degree* attribute. DEG1 only picks from the most important scale degrees: the tonic, third, fifth, and octave. DEG2 picks from the remaining degrees. Finally, PR can be used within a pipe to apply an optional step f with probability p. It will be used throughout this section.

CUT : $(s \in \mathbb{R}_\lambda) \mapsto trim(choice(split([size_{set}(s), \langle\rangle]), split([\langle\rangle, size_{set}(s)])))$
REP : $(s \in \mathbb{R}_\lambda) \mapsto trim(repeat_{\lceil}(size_{set}(s), index_{set}))$
DEG1 : $degree_{set}(choice(I, III, V, VIII))$
DEG2 : $degree_{set}(choice(II, IV, VI, VII))$
PR : $(p \in [0,1]_\lambda, f : V_\lambda^*) \mapsto \langle rand \leq p \longrightarrow f \rangle$

2.1.2 Example: Bass Motif

BASS generates notes for a bass line and is the first and simplest of our four motif generators. We start by repeating a random time interval. Since the primary purpose of the bass is to communicate the fundamental frequency, we will only change the degree if it is not the first note of the motif. Even then we only pick from the most important scale degrees as provided by DEG1. In the last two steps we select a random octave and calculate the corresponding frequency. A subset of the possible results is shown in Figure 1.

BASS : $\langle$
$\quad$ REP$(choice(measure, whole, half, quarter))$
$\quad$ $index > 0 \longrightarrow$ DEG1
$\quad$ $octave_{set}(choice(1, 2))$
$\quad$ $freq_{set}(diatonic)$
$\rangle$

2.1.3 Example: Pad Motif

PAD generates chord notes in order to provide harmonic context for the piece. Per default the chord will be a triad, but can be extended to a seventh chord

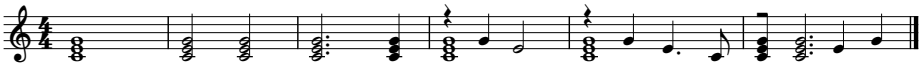

Figure 2: Each measure shows a possible result of the PAD grammar (File: pad.mp3). We normalized the results to C-major and common time, and use $chord = 3$. The splitting of the individual chord notes in the last three examples causes a high degree of polyphony, which is difficult to represent in a single staff.

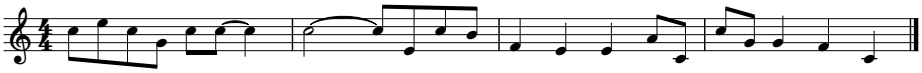

Figure 3: Each measure shows a possible result of the LEAD grammar lead.mp3). We normalized the results to C-major and common time.

or higher by setting the *chord* attribute. The factor $2i$ calculates scale degrees that correspond to chord notes. For example, if $chord = 3$, the resulting degrees will be (I, III, V). For additional texture we randomly divide the scope using CUT, which splits off a half, quarter, or eighth note from the start or end of the interval. We apply it once before the chord note generation, which splits the whole chord, and once after, which splits each note individually. Unlike BASS, we determine the octave for the whole motif and not for each note individually. A subset of the possible results is shown in Figure 2.

PAD : $\langle$
$octave_{set}(choice(3, 4))$
$\mathrm{PR}(0.5, \mathrm{CUT}(choice(half, quarter, eighth)))$
$range(chord_{or}(3), i \mapsto degree_{set}(2i))$
$\mathrm{PR}(0.5, \mathrm{CUT}(choice(half, quarter, eighth)))$
$freq_{set}(diatonic)$
$\rangle$

2.1.4 Example: Lead Motif

LEAD is the most complex of the four motif grammars as it will generate the melody, which should ideally be distinct in its movement and rhythm. We start by applying two optional splits and assign a random degree with DEG1 to generate a preliminary structure for the motif. An optional REP further divides each note and the degree will be reassigned with a certain probability. Note that we use DEG2 with a lower probability, as it uses degrees that are less harmonic than those of DEG1. A subset of the possible results is shown in Figure 3 and Figure 4.

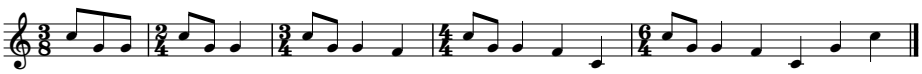

Figure 4: These examples show how the LEAD grammar behaves under varying metric constraints (File: lead.meter.mp3). The seed is constant for each measure.

LEAD : ⟨
PR(0.5, CUT($choice(half, quarter, eighth)$))
PR(0.5, CUT($choice(half, quarter, eighth)$))
DEG1
PR(0.5, REP($choice(half, quarter, eighth)$))
PR(0.5, DEG1)
PR(0.25, DEG2)
$freq_{set}(diatonic)$
⟩

2.1.5 Example: Drum Motif

DRUM is our final motif grammar and generates regular pulses with a randomly selected interval. As most percussion instruments do not have pitch, we will instead modify the gain attribute to increase the variability. We scale the gain relative to the duration of the entity and then alter the gain of every second beat to increase the fidelity. In the last step we add a random offset to beats, as long as they are equal or less than a quarter in length. A subset of the possible results is shown in Figure 5.

DRUM : ⟨
REP($choice(measure, half, quarter, eighth)$)
$gain_{set}(gain * span/whole)$
$\langle odd(index) \longrightarrow gain_{set}(gain/2)\rangle$
$\langle span \leq quarter \longrightarrow time_{set}(time + choice(0, span/2))\rangle$
⟩

2.1.6 Example: Motif Selection

Finally, we wrap the four motif generators in a single fork and use the nominal *role* attribute to select only one of the branches. We will later assign a role to each voice, which will cause them to play motifs in a consistent style. The

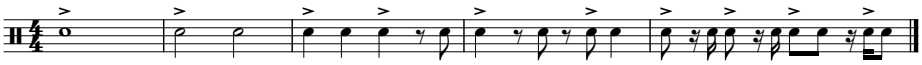

Figure 5: Each measure shows a possible result of the DRUM grammar (File: drum.mp3). We normalized the results to common time. The accents indicate that the notes with an even index are louder than the rest. The scaling of the loudness in relation to the duration is not depicted.

set of roles is easily extensible with additional grammars. We can even have multiple motif grammars for a role by wrapping them in the *choice* function, for example *choice*(DRUM1, DRUM2). A motif, as we have defined it earlier, has no higher-level organization. Our next step is the generation of a musical layout that contains repetition. The leaves of this structure can then be filled by the MOTIF grammar.

MOTIF : [
$role$ = 'lead' $\longrightarrow$ LEAD
$role$ = 'pad' $\longrightarrow$ PAD
$role$ = 'bass' $\longrightarrow$ BASS
$role$ = 'drum' $\longrightarrow$ DRUM
]

2.2 Layouts

The next step is the integration of the motifs into a larger structure. As the motif grammars generate a large variety of possible arrangements, repetition must be provided in the larger context. In the context of this example we define a musical layout as a sequential arrangement of entities, where each entity is associated with a nominal *label* attribute that identifies the shape of the subtree. For example, a binary form with repeated themes can be described by four labels AABB. We will generate such patterns with a binary tree generator LAYOUT, which randomly repeats nodes using the SYNC helper function defined below. We will then synchronize multiple layouts, one for each voice, using the LAYER grammar. An optional application of the LABEL grammar allows us to select individual seeds for the motifs and generate sparse layouts by dropping measures.

2.2.1 Example: Probabilistic Synchronization

A simple grammar over a vocabulary of labels $\{A, B, \ldots\}$ and rules such as $\{A \longrightarrow AA, A \longrightarrow AB, \ldots\}$ could be used to generate patterns with repetitions. We consider two cases: (1) the entities on the RHS are synchronized (AA, BB), (2) or they are not synchronized (AB, BA). We isolate this choice in a helper function SYNC, which takes a synchronization probability p and a

sentence expression f that generates the subtree roots. We define the attribute *sync*, which will be used as a temporary variable that holds a random number. Depending on the probability p, we will either use *sync* as the seed for each subtree, or use a random value. In the first case, the result will be completely monotonous (AAA...), and in the second case it will have no repetition at all (ABC...). SYNC on its own is not yet very useful, but this will change once we apply it on multiple levels of the derivation tree.

$$
\begin{aligned}
&\text{SYNC} : (p \in [0,1]_\lambda, f \in V_\lambda^*) \mapsto \langle \\
&\quad sync_{set}(rand) \\
&\quad f \\
&\quad \langle p < sync \longrightarrow sync_{set}(rand) \rangle \\
&\quad seed(sync) \\
&\rangle
\end{aligned}
$$

2.2.2 Example: Recursive Layout Generation

LAYOUT recursively divides the input entity and applies SYNC at each branch point. We use the *derive* function to express a recursion and a new attribute *depth* for terminating the derivation after a certain number of levels. Since we use the constant 2 for the splits, the result is a binary tree, and the total length of the pattern is 2^n, where n is the initial depth. The *monotony* attribute can be used to control the synchronization probability, and its effect is visualized in Figure 6. For example, with an initial depth of two, the rule will be applied once at the root and once for each of the resulting parts, yielding four entities in total. If only the first split is synchronized, the grammar generates ABAB. If all but the first split get synchronized, the pattern is AABB. With LAYOUT we can already generate interesting pieces of variable length by simply feeding its result to the motif generators. For this example we assume f is the identity function $\langle\rangle$. In the next example we will use the f parameter to further modify the split results, allowing us to synchronize layouts.

$$
\begin{aligned}
&\text{LAYOUT} : (f \in V_\lambda^*) \mapsto derive(\\
&\quad depth_! > 0 \longrightarrow \langle \\
&\qquad depth^!_{set}(depth - 1) \\
&\qquad \text{SYNC}(monotony_{or}(0), split(range(2))) \\
&\qquad f \\
&\quad \rangle \\
&)
\end{aligned}
$$

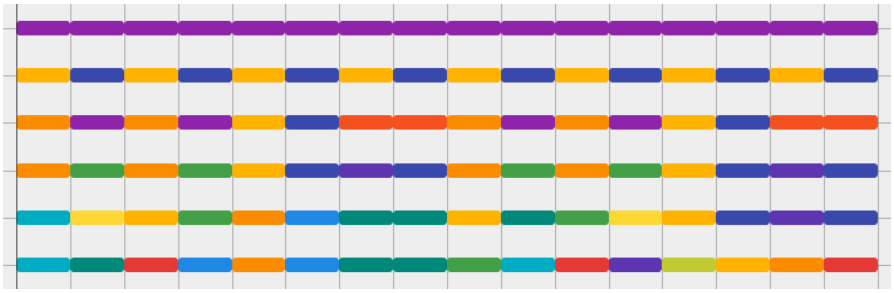

Figure 6: This visualization shows six generations of LAYOUT with *depth* = 4, where the monotony decreases from top (*monotony* = 1) to bottom (*monotony* = 0) in steps of 0.2. Note that the visualization is limited to 16 colors, which causes the last row to show random repetitions, even though the monotony is zero and no synchronization occurred.

2.2.3 Example: Synchronization between Layouts

While LAYOUT could be used to generate the structure of a piece, the use of a single layout will synchronize the motifs across all voices. This can be desirable. For example, in a repeated binary form with the layout AABB we generate the A motif twice in all voices, then the B motif twice in all voices, leading to two pairs of perfect repetitions. But for the sake of variability it would be interesting if one voice could play AABB, another ABAB, and yet another AAAB. This can be achieved by generating a layout for each voice individually. Then again, this will lead to the voices to be perfectly desynchronized, which is interesting but may sound chaotic over longer pieces.

As a generalization of the two strategies, we define the LAYER grammar, which allows us to gradually adjust the synchronization between multiple layouts. We assume that the *layer* attribute contains a number that uniquely identifies the layout, or alternatively use a random number. Only then do we synchronize the generation of the layout using a global seed stored in the *piece* attribute. We define the *diversity* attribute, which is the probability that one branch of a layout gets desynchronized from other instances of LAYOUT that were generated with the same *piece* value. The idea is that we can generate the layout individually for each voice, but use the same *piece* value for all of them. With a diversity of zero, the result is effectively the same as if we had generated a single layout for all voices. If the diversity is one, all voices will have a unique layout, as only the unique seed in *layer* will be used. The effect of the *diversity* attribute is visualized in Figure 7.

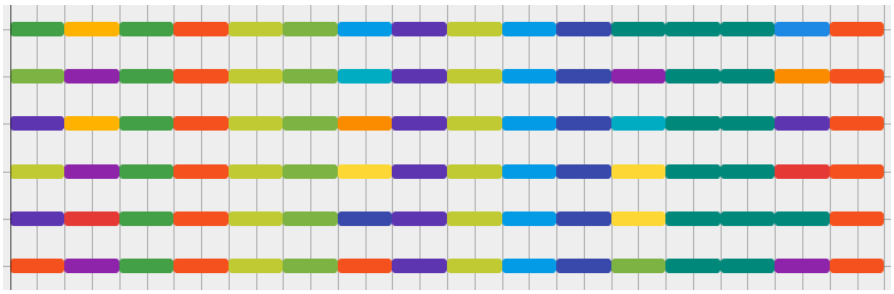

Figure 7: This visualization shows six generations of LAYER with $depth = 4$, $diversity = 0.25$, and a different value for the *layer* attribute in each row.

LAYER : ⟨
$layer_{or}(rand)$
$seed(piece)$
LAYOUT(PR($diversity, seed(rand + layer)$))
⟩

2.2.4 Example: Layout Transfer Function

At last, we use LABEL as a transfer function for mapping the random numbers generated by LAYOUT to a set of predefined labels. To achieve this we set the *label* attribute with a random element from a numeric sequence stored in the *labels* attribute. This gives us greater control over the resulting layout, as we can change individual labels, while keeping everything else as it was. Another aspect that we control with the labels is *sparsity*. A voice does not have to cover the whole piece. In fact, the piece will sound much more interesting if certain voices only appear during certain parts of the piece. We use the label 0 as a marker for entities that should be removed. For example, if $labels = (0, 1, 2)$, on average a third of the entities will be removed. The effect of various settings for *labels* is shown in Figure 8. As a final step, we set the seed by combining *label* with the global *piece* seed. It is important that we add a global source of entropy, otherwise we will restrict the results to the number of unique values in the label pool.

LABEL : ⟨
$label_{set}(choice(labels))$
$\langle label = 0 \longrightarrow [] \rangle$
$seed(piece + label)$
⟩

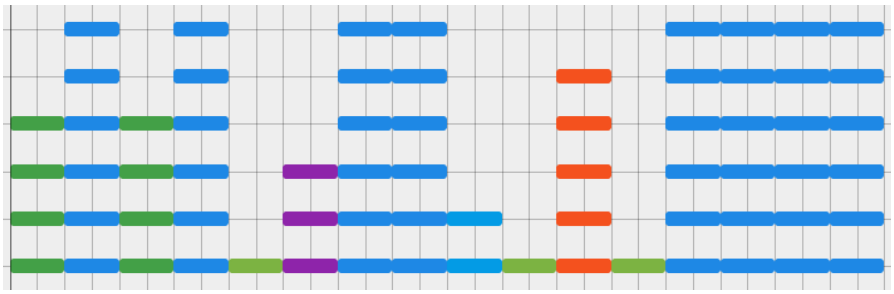

Figure 8: This visualization shows six generations of LAYER with $depth = 4$, using $labels = (1, 0, 0, 0, 0, 0)$ for the first instance, $labels = (1, 2, 0, 0, 0, 0)$ for the second, up to $labels = (1, 2, 3, 4, 5, 6)$ for the last one. The sparsity of the layer increases with the number of zeros in the label pool.

2.3 Chords

With the MOTIF and LAYER grammars we can already generate a wealth of pieces, but they will all sound quite monotonous. What is missing is a chord progression. We do not attempt to approximate any particular harmonic practice. Given that both grammars are context-free we should be able to define them within our framework, but we will not attempt this here. Instead, our progression example will be rather primitive: we first generate a chord pool with the CHORDS grammar, and after generating a temporal division we assign a random chord from the pool to each entity. This process is defined in the PROGRESSION grammar. We model the harmonic information of a chord with the attributes $K_H = \{key, mode, harmony, chord\}$, where *chord* encodes the number of chord factors, for example 3 for a triad, 4 for a seventh chord, and so forth. In addition we define the *chords* attribute over the sentence V^*, in which we will store the chord pool. On the piece level we will use it to store the complete harmonic progression, so that we can query it with the CHORD grammar and use the same chords in every voice.

2.3.1 Example: Chord Pool Generation

CHORDS generates the chord pool. It does not matter how the chords are arranged. We generate four chords, the tonic (I), subdominant (IV), dominant (V), and parallel minor (VI) in one of two modes: $major = I$ or $minor = VI$. Each chord will be a triad, except for the dominant, where we add the seventh by setting *chord* to 4. We could additionally alter the key, or use Jazz chords with added sevenths and ninths. Since the chords will be arranged randomly it is essential to encode harmonic constraints already in the chord pool, for example by not excessively mixing keys or modes.

$$
\begin{aligned}
&\text{CHORDS} : \langle \\
&\quad mode_{set}(choice(major, minor)) \\
&\quad chord_{set}(3) \\
&\quad [harmony_{set}(I), harmony_{set}(IV), harmony_{set}(V), harmony_{set}(VI)] \\
&\quad \langle harmony = V \longrightarrow chord_{set}(4) \rangle \\
&\rangle
\end{aligned}
$$

2.3.2 Example: Random Chord Progressions

PROGRESSION generates a chord progression for a sentence f using our chord pool. We first set the *chords* attribute with the result of the CHORDS grammar. Next, we apply f, which generates the temporal division for our progression. Finally, for each entity generated by f, we select a random chord from *chords* and append only the harmonic attributes K_H.

$$
\begin{aligned}
&\text{PROGRESSION} : (f \in V_\lambda^*) \mapsto \langle \\
&\quad chords_{set}(\text{CHORDS}) \\
&\quad f \\
&\quad append(\langle choice(chords), select(K_H) \rangle) \\
&\rangle
\end{aligned}
$$

2.3.3 Example: Chord Texture Query

CHORD is a helper function that we will use to query the *chords* attribute. Since *query* returns all overlapping entities, we use a function $first$ that returns only the first entity of a sentence. In order to guarantee that there is at least one entity, we use the identity function $\langle\rangle$ as a fallback. Since we want to preserve attributes on the input entity, we again only append the harmonic attributes K_H.

$$
\begin{aligned}
&\text{CHORD} : append(\langle \\
&\quad first([query(chords), \langle\rangle]) \\
&\quad select(K_H) \\
&\rangle)
\end{aligned}
$$

2.4 Pieces

We now have all the necessary components for generating a complete, polyphonic piece. Our piece will consist of multiple voices in different roles, which we define in the VOICE grammar. The harmonic information on which we build these

voices comes from a singular chord progression layer that we generate with the PROGRESSION grammar. The GLOBALS grammar will randomize global parameters on the axiom, and the PIECE grammar will integrate all these components and apply them in the correct order.

2.4.1 Example: Voices

VOICES generates the roots for the voice layers of our polyphonic piece. We define a *role* for each voice, which will be used by MOTIF to select one of the motif grammars. The *play* attribute is the name for an instrument or instrument family. How this value is interpreted is up to playback system. For the pad, bass, and drums we use the label pool $(1, 2)$, which means each has at most two different motifs per piece. The label pool for the two leading voices is more interesting. When the first voice plays motif 1, the second voice is silent, as indicated by the label 0. When the second voice plays motif 2, the first voice is silent. Motif 3 will be played by both at the same time. This only works if we additionally set the *layer* attribute of the lead voices to the same value.

VOICES : [

$\langle role_{set}(\text{'bass'}), play_{set}(\text{'contrabass'}), labels_{set}(1, 2)\rangle$

$\langle role_{set}(\text{'pad'}), play_{set}(\text{'piano'}), labels_{set}(1, 2)\rangle$

$\langle role_{set}(\text{'lead'}), play_{set}(\text{'violin'}), labels_{set}(1, 0, 3), layer_{set}(1)\rangle$

$\langle role_{set}(\text{'lead'}), play_{set}(\text{'piccolo'}), labels_{set}(0, 2, 3), layer_{set}(1)\rangle$

$\langle range(4), role_{set}(\text{'drum'}), play_{set}(\text{'percussion'}), labels_{set}(1, 2)\rangle$

]

2.4.2 Example: Randomizing Global Attributes

GLOBALS initializes the attributes that will be constant for the whole piece. We start by setting the global attributes and the *piece* attribute, which will be used by our LAYER and LABEL grammars. Our motif generators are flexible enough that they can handle almost any meter, as shown in Figure 4. There is no particular reason for setting exactly these attributes either. For example, one could use a different *tempo* for different parts of a piece, or we could set the *monotony* for each voice individually. Alternatively, we can integrate these attributes as parameters in a graphical interface to give users primitive control over the generated result, which we show in Figure 9. For any other canonical attributes that are missing here, we assume the default values that were established in this chapter.

GLOBALS : ⟨

$piece_{set}(rand)$

$key_{set}(\lfloor uniform(0, 12) \rfloor)$

$tempo_{set}(uniform(80, 140))$

$beats_{set}(choice(2, 3, 4, 6))$

$beatType_{set}(choice(4, 8))$

$monotony_{set}(uniform(0.25, 0.75))$

$diversity_{set}(uniform(0, 0.5))$

$depth_{set}(choice(2, 3, 4))$

⟩

2.4.3 Example: Piece

Finally, we define the PIECE grammar, which will generate the complete piece. For the overall duration of the piece we set *span* to a number of measures equal to the number of leaves in the layer. For the *chords* attribute we use the PROGRESSION grammar and pass the LAYER as its argument. This means the chord progression will have the same temporal structure as the voices, which is useful since we will later query it with the CHORD function. The last rule applies our various sub-grammars in sequence. Note that the LABEL and CHORD sub-grammars generate only one entity, while VOICES, LAYER, and MOTIF may generate any number of entities.

PIECE : ⟨

GLOBALS

$span_{set}(2^{depth} * measure)$

$chords_{set}$(PROGRESSION(LAYER))

⟨VOICES, LAYER, LABEL, CHORD, MOTIF⟩

⟩

Figure 10 shows one example of the PIECE grammar. We can see how the algorithm reuses the limited set of motifs and how the two lead voices interact. The piano grammar has more variation than the other voices, as the seventh chords have an additional note. The piece lacks overall structure. For example, there is no satisfying harmonic resolution at the end, but this would be a lot to ask from our random progression generator. Nevertheless, the results are quite satisfying. Most importantly, the piece features polyphony on the large scale as a set of semi-independent voices, which are all synchronized to a hidden chord layer, and on the small scale in the form of chords and random arpeggios. We were able to achieve this within the boundaries of our theoretic framework,

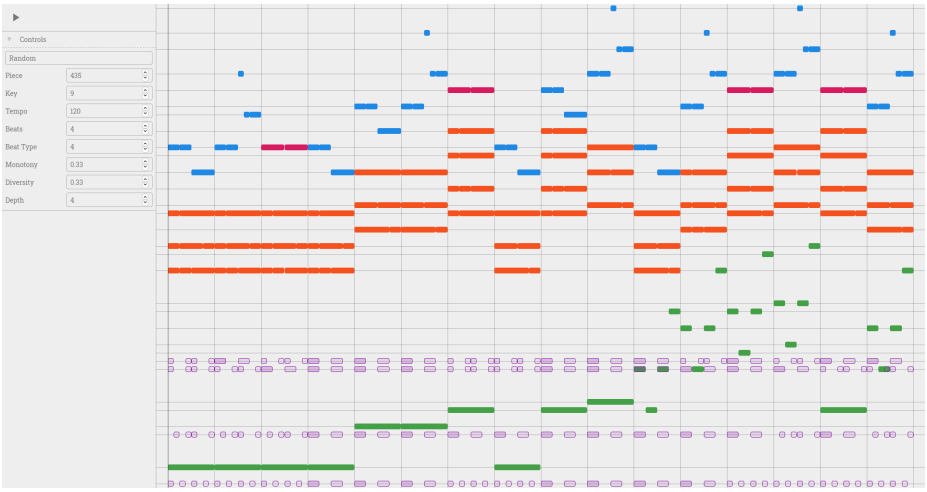

Figure 9: This screenshot shows the interface for one result of the generator (File: piece1.mp3). The colors in the visualization indicate the instrument. Some parts of the magenta voice are hidden by the blue voice. The controls on the left side correspond to the attributes set by the GLOBALS sub-grammar.

without compromising musical quality or our intuitions about the composition process.

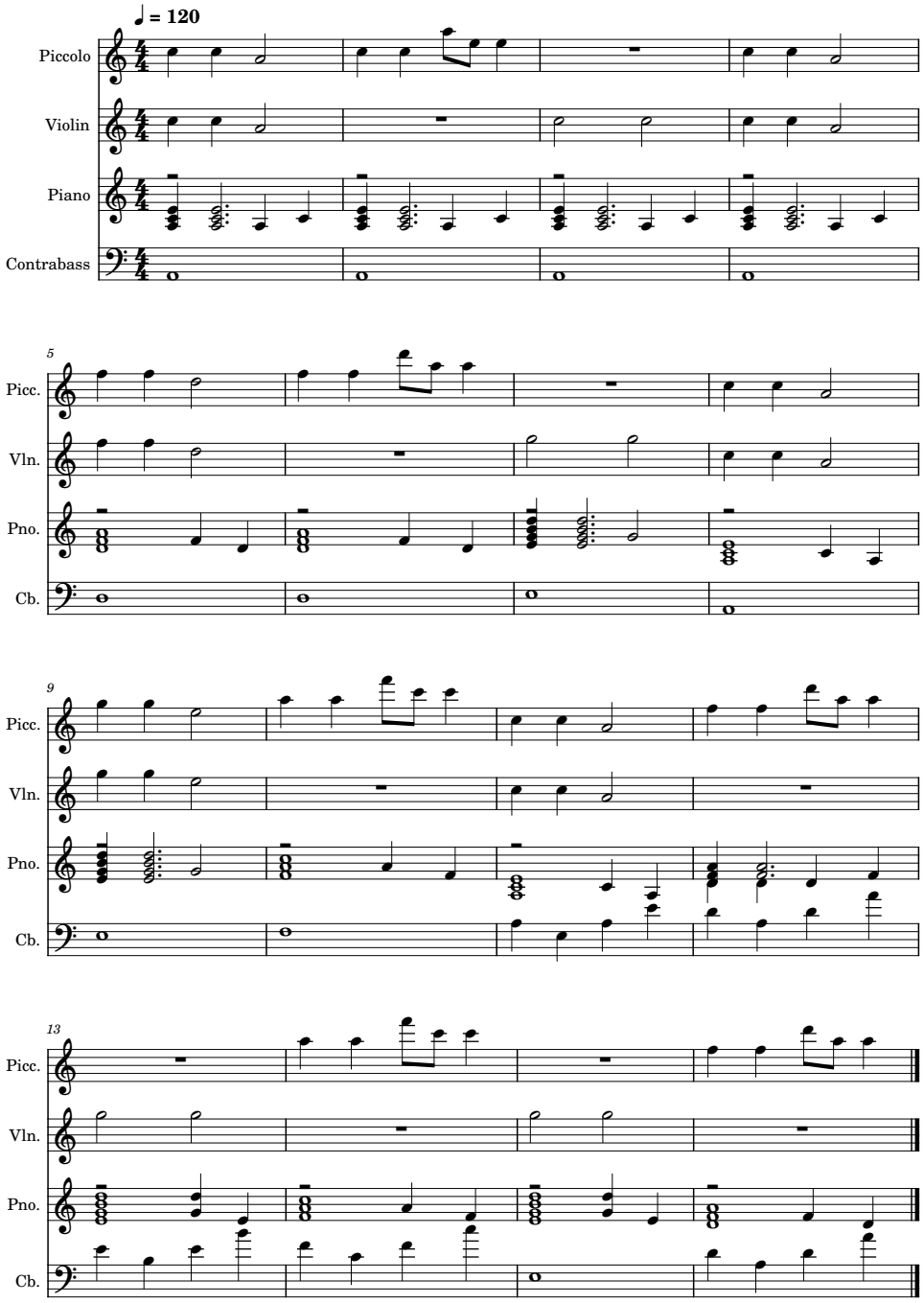

Figure 10: Manual transcription of one generation of the PIECE grammar. The drum voices are not included. The document attachments include two musical interpretations, one with percussion (File: piece1.mp3) and one without (File: piece1.score.mp3).

A Appendix

This section contains the specification for the music generator described in Section 2. Note that only the grammar and GUI definitions are included, not the complete library.

A.1 Grammar Definition

This section lists the theoretic code from Section 2. Listing it here in compact form allows for convenient comparison with the corresponding TypeScript source code in Section A.2.

$\text{CUT} : (s \in \mathbb{R}_\lambda) \mapsto trim(choice(split([size_{set}(s), \langle\rangle]), split([\langle\rangle, size_{set}(s)])))$
$\text{REP} : (s \in \mathbb{R}_\lambda) \mapsto trim(repeat_{\lceil}(size_{set}(s), index_{set}))$
$\text{DEG1} : degree_{set}(choice(I, III, V, VIII))$
$\text{DEG2} : degree_{set}(choice(II, IV, VI, VII))$
$\text{PR} : (p \in [0,1]_\lambda, f : V_\lambda^*) \mapsto \langle rand \leq p \longrightarrow f \rangle$

$\text{BASS} : \langle$
    $\text{REP}(choice(measure, whole, half, quarter))$
    $index > 0 \longrightarrow \text{DEG1}$
    $octave_{set}(choice(1, 2))$
    $freq_{set}(diatonic)$
$\rangle$

$\text{PAD} : \langle$
    $octave_{set}(choice(3, 4))$
    $\text{PR}(0.5, \text{CUT}(choice(half, quarter, eighth)))$
    $range(chord_{or}(3), i \mapsto degree_{set}(2i))$
    $\text{PR}(0.5, \text{CUT}(choice(half, quarter, eighth)))$
    $freq_{set}(diatonic)$
$\rangle$

$\text{LEAD} : \langle$
    $\text{PR}(0.5, \text{CUT}(choice(half, quarter, eighth))$
    $\text{PR}(0.5, \text{CUT}(choice(half, quarter, eighth))$
    DEG1
    $\text{PR}(0.5, \text{REP}(choice(half, quarter, eighth))$
    $\text{PR}(0.5, \text{DEG1})$
    $\text{PR}(0.25, \text{DEG2})$
    $freq_{set}(diatonic)$
$\rangle$

$\text{DRUM} : \langle$
    $\text{REP}(choice(measure, half, quarter, eighth))$
    $gain_{set}(gain * span/whole)$
    $\langle odd(index) \longrightarrow gain_{set}(gain/2) \rangle$
    $\langle span \leq quarter \longrightarrow time_{set}(time + choice(0, span/2)) \rangle$
$\rangle$

MOTIF : [
    $role$ = 'lead' $\longrightarrow$ LEAD
    $role$ = 'pad' $\longrightarrow$ PAD
    $role$ = 'bass' $\longrightarrow$ BASS
    $role$ = 'drum' $\longrightarrow$ DRUM
]

SYNC : $(p \in [0,1]_\lambda, f \in V_\lambda^*) \mapsto \langle$
    $sync_{set}(rand)$
    f
    $\langle p < sync \longrightarrow sync_{set}(rand) \rangle$
    $seed(sync)$
$\rangle$

LAYOUT : $(f \in V_\lambda^*) \mapsto derive($
    $depth_! > 0 \longrightarrow \langle$
        $depth^!_{set}(depth - 1)$
        SYNC$(monotony_{or}(0), split(range(2)))$
        f
    $\rangle$
)

LAYER : $\langle$
    $layer_{or}(rand)$
    $seed(piece)$
    LAYOUT(PR$(diversity, seed(rand + layer))$)
$\rangle$

LABEL : $\langle$
    $label_{set}(choice(labels))$
    $\langle label = 0 \longrightarrow [] \rangle$
    $seed(piece + label)$
$\rangle$

CHORDS : $\langle$
    $mode_{set}(choice(major, minor))$
    $chord_{set}(3)$
    $[harmony_{set}(I), harmony_{set}(IV), harmony_{set}(V), harmony_{set}(VI)]$
    $\langle harmony = V \longrightarrow chord_{set}(4) \rangle$
$\rangle$

PROGRESSION : $(f \in V_\lambda^*) \mapsto \langle$
    $chords_{set}$(CHORDS)
    f
    $append(\langle choice(chords), select(K_H) \rangle)$
$\rangle$

CHORD : $append(\langle$

$first([query(chords), \langle\rangle])$
$select(K_H)$
$\rangle)$

VOICES : [
$\langle role_{set}(\text{'bass'}), play_{set}(\text{'contrabass'}), labels_{set}(1, 2)\rangle$
$\langle role_{set}(\text{'pad'}), play_{set}(\text{'piano'}), labels_{set}(1, 2)\rangle$
$\langle role_{set}(\text{'lead'}), play_{set}(\text{'violin'}), labels_{set}(1, 0, 3), layer_{set}(1)\rangle$
$\langle role_{set}(\text{'lead'}), play_{set}(\text{'piccolo'}), labels_{set}(0, 2, 3), layer_{set}(1)\rangle$
$\langle range(4), role_{set}(\text{'drum'}), play_{set}(\text{'percussion'}), labels_{set}(1, 2)\rangle$
]

GLOBALS : $\langle$
$piece_{set}(rand)$
$key_{set}(\lfloor uniform(0, 12)\rfloor)$
$tempo_{set}(uniform(80, 140))$
$beats_{set}(choice(2, 3, 4, 6))$
$beatType_{set}(choice(4, 8))$
$monotony_{set}(uniform(0.25, 0.75))$
$diversity_{set}(uniform(0, 0.5))$
$depth_{set}(choice(2, 3, 4))$
$\rangle$

PIECE : $\langle$
GLOBALS
$span_{set}(2^{depth} * measure)$
$chords_{set}(\text{PROGRESSION(LAYER)})$
$\langle$VOICES, LAYER, LABEL, CHORD, MOTIF$\rangle$
$\rangle$

A.2 Grammar Definition (TypeScript)

This section lists the TypeScript source code that corresponds to the grammar in Section 2. The import statements at the beginning of the file load the library functions provided by our framework. The `'./drums'` module defines constants for a non-standard percussion instrument and is not part of the library. The typed declaration `MyEntity` extends the `Entity` type with custom attributes. The rest of the file closely resembles the code in Section A.1.

```
001 import {diatonic, eighth, Entity, gleitz, half, measure, measures, overlaps,
        quarter, rel, repeat, split, translate, trim, whole} from '@dipl/lib-music'
002 import {clone, Exp, filter, first, fork, Fun, indexed, map, noop, pipe, pipeDeep,
        range, Seq, set, setx, use, val, when} from '@dipl/lib-core'
003 import {choice, pr, rand, seed, uniform} from '@dipl/lib-rng'
004 import {frac, lte, pow, round} from '@dipl/lib-math'
005 import {acousticGrandPiano, contrabass, I, II, III, IV, major, minor, musyngKite,
```

```
        piccolo, V, VI, VII, VIII, violin} from '@dipl/lib-constants'
import * as drums from './drums'

export type MyEntity = Entity & {
  piece: number
  depth: number
  monotony: number
  diversity: number
  layer: number
  label: number
  labels: number[]
  chord: number
  role: 'bass' | 'lead' | 'pad' | 'drum'
  chords: MyEntity[]
}

function Cut<X extends Entity> (size: Exp<number, X>): Fun<X[], X> {
  return trim(choice(
    split<X>(setx({size}), noop),
    split<X>(noop, setx({size}))
  ))
}

function Rep<X extends Entity> (size: Exp<number, X>): Fun<X[], X> {
  return trim(indexed(repeat<X>({
    size,
    cover: true
  }), i => set({index: i})))
}

function Deg1<X extends Entity> (): Fun<X, X> {
  return setx({degree: choice(I, III, V, VIII)})
}

function Deg2<X extends Entity> (): Fun<X, X> {
  return setx({degree: choice(II, IV, VI, VII)})
}

function Pr<X extends Entity> (p: Exp<number, X>, f: Exp<Seq<X>, X>): Fun<X[], X> {
  return when(pr(p), f)
}

function Bass<X extends Entity> (): Fun<X[], X> {
  return pipe(
    Rep(choice<number, X>(half, quarter)),
    when(x => x.index > 0, Deg1()),
```

```
    setx({octave: choice(1, 2)}),
    setx({freq: diatonic})
  )
}

function Pad<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    setx({octave: choice(3, 4)}),
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    range(x => x.chord || 3, i => set({degree: i * 2})),
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    setx({freq: diatonic})
  )
}

function Lead<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    Pr(0.5, Cut(choice<number, X>(half, quarter, eighth))),
    Deg1(),
    Pr(0.5, Rep(choice<number, X>(half, quarter, eighth))),
    Pr(0.5, Deg1()),
    Pr(0.25, Deg2()),
    setx({freq: diatonic})
  )
}

function Drum<X extends Entity> (): Fun<X[], X> {
  return pipe<X>(
    Rep(choice<number, X>(measure, half, quarter, eighth)),
    setx({gain: x => x.gain * x.span / val(whole, x)}),
    x => when(x.index % 2 === 1, setx({gain: x.gain / 2})),
    x => when(lte(x.span, quarter), translate<X>(choice(0, rel(1 / 2))))
  )
}

function Motif<X extends MyEntity> (): Fun<X[], X> {
  return trim((x: X) => ({
    lead: Lead<X>(),
    pad: Pad<X>(),
    bass: Bass<X>(),
    drum: Drum<X>()
  }[x.role]))
}

function Sync<X extends Entity> (p: Exp<number, X>, f: Exp<X[], X>): Fun<X[], X> {
```

```
  return use(rand, sync => pipe<X>(
    f,
    use(p, p => seed<X>(p < sync ? rand : sync))
  ))
}

function Layout<X extends MyEntity> (f: Exp<X[], X>): Fun<X[], X> {
  return pipeDeep(
    x => x.depth,
    Sync(x => x.monotony, split(clone(2))),
    f
  )
}

function Layer<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    seed(x => x.piece),
    Layout(when(pr(x => x.diversity), seed<X>(rand, x => x.layer)))
  )
}

function Label<X extends MyEntity> (): Fun<X[], X> {
  return pipe(
    x => setx({label: choice(...x.labels)}),
    x => when<X>(x.label === 0, []),
    x => seed(x.piece, x.label, x.layer)
  )
}

function Chords<X extends Entity> (): Fun<X[], X> {
  return pipe(
    setx({mode: choice(major, minor)}),
    set({chord: 3}),
    map([I, IV, V, VI], harmony => set({harmony})),
    when(x => x.harmony === V, set({chord: 4}))
  )
}

function Progression<X extends MyEntity> (f: Exp<X[], X>): Fun<X[], X> {
  return pipe<X>(
    setx({chords: Chords}),
    f,
    x => ChordAppend(choice(...x.chords))
  )
}
```

```
function ChordAppend<X extends Entity> (c: Exp<X, X>): Fun<X, X> {
  return use(c, ({harmony, key, mode, chord}) => set(
    {harmony, key, mode, chord}
  ))
}

function Chord<X extends MyEntity> (): Fun<X, X> {
  return ChordAppend<X>(
    first(filter<MyEntity, X>(x => x.chords, overlaps), noop())
  )
}

function Voices<X extends MyEntity> (): Fun<X[], X> {
  const Play = (n: number) => gleitz(musyngKite, n)
  return fork(
    setx({
      role: 'pad',
      play: Play(acousticGrandPiano),
      labels: [1, 2],
      color: 2,
      layer: rand
    }),
    setx({
      role: 'lead',
      play: Play(violin),
      labels: [1, 0, 3],
      color: 0,
      layer: 1
    }),
    setx({
      role: 'lead',
      play: Play(piccolo),
      labels: [0, 2, 3],
      color: 1,
      layer: 1
    }),
    setx({
      role: 'bass',
      play: Play(contrabass),
      labels: [1, 2],
      color: 3,
      layer: rand
    }),
    pipe(
      clone(4),
      seed<X>(rand),
```

```
      setx({
        role: 'drum',
        play: drums.font,
        freq: choice(...drums.hats, ...drums.snares, ...drums.snares),
        labels: [1, 2],
        gain: 1 / 2,
        color: 4,
        layer: rand
      })
    )
  )
}

export function Globals<X extends MyEntity> (): Fun<X, X> {
  return setx({
    piece: round(uniform(0, 10000)),
    key: round(uniform(0, 12)),
    tempo: round(uniform(80, 140)),
    beats: choice(2, 3, 4, 6),
    beatType: choice(4, 8),
    monotony: frac(round(uniform(25, 75)), 100),
    diversity: frac(round(uniform(0, 50)), 100),
    depth: choice(2, 3, 4)
  })
}

export function Init<X extends MyEntity> (f: Fun<X[], X>): Fun<X[], X> {
  return pipe(
    x => seed(x.piece),
    setx({span: measures(pow(2, x => x.depth))}),
    f
  )
}

export function Piece<X extends MyEntity> (): Fun<X[], X> {
  return Init(pipe<X>(
    setx({chords: Progression(Layer)}),
    Voices,
    Layer,
    Label,
    Chord,
    Motif
  ))
}
```

A.3 GUI Definition (TypeScript)

This section lists the TypeScript source code that defines the graphical user interface shown in Figure 9. We assume that the grammar definition code from Section A.2 is located in the module `'./grammar'`. We define the starting entity, the visual controls, and the grid lines for the score visualization. The `vis` function generates the interface, and the `render` function attaches it to the browser window.

```
01  import {render} from 'preact'
02  import {fieldset, inputs, label, number, presets, propKey, vis} from '@dipl/dom-vis'
03  import {EntityDefaults, measure, repeat} from '@dipl/lib-music'
04  import {fork, Fun, map, set} from '@dipl/lib-core'
05  import {Globals, Init, MyEntity, Piece} from './grammar'

07  const entity: MyEntity = {
08    ...EntityDefaults,
09    piece: Math.round(Math.random() * 10000),
10    depth: 4,
11    monotony: 0.5,
12    diversity: 0.5,
13    layer: 0,
14    label: 0,
15    labels: [1],
16    chord: 3,
17    role: 'drum',
18    chords: []
19  }

21  const input = inputs<MyEntity>(
22    fieldset('Controls', inputs(
23      presets({random: Globals}),
24      propKey(label, 'piece', number(0, 10000, 1)),
25      propKey(label, 'key', number(0, 12)),
26      propKey(label, 'tempo', number(80, 140)),
27      propKey(label, 'beats', number(2, 6)),
28      propKey(label, 'beatType', number(2, 8)),
29      propKey(label, 'monotony', number(0, 1, 0.01)),
30      propKey(label, 'diversity', number(0, 1, 0.01)),
31      propKey(label, 'depth', number(0, 5))
32    ))
33  )

35  function Lines<X extends MyEntity> (notes: X[]): Fun<X[], X> {
36    return Init(
37      fork(
```

```
      repeat<X>({size: measure}),
      map(notes, x => set({freq: x.freq}))
    )
  )
}

render(
  vis({
    input,
    axiom: entity,
    notes: Piece,
    lines: Lines
  }),
  document.body
)
```