Ruled Random Generation

Valérian Pittet

January 20, 2015

Abstract

This project builds a programming interface to randomly generate music according to user-defined grammars. To reach this aim, it provides a model to abstract differents components of a melody a small embedded domain specific language to describe grammars, a (weak) generation algorithm, the ability to trigger a special generation after specified event and a midi interface based on javax.sound.midi to output the results.

The current report is divided in three parts. The first one presents the model, how melodies are generated and the key point of implementation. The second one shows an incremental process of using provided features to tune a good generation. The last one gives a subjective review of the project and discusses what could be improved.

Contents

1 Introduction

When designing this project, the first aim was to use a musical model that clearly separates the tonal aspect (heights of notes) and the rhythm. Having this kind of separation makes it very simple to radically change a generated slow melody into a quick and energetic one.

The next step was to use an harmonic progression as the backbone of the melody. A ponctual constraint ensuring that somtimes, the melody follows some harmonic rules. Given the aim of ruled random generation, the grammars, well-known tool to generate controlled sequences, were naturally chosen as basic representation. However, the result : a set of parameters modeling a music generated using grammars, was not satisfying.

Indeed it lacks the ability to produce specific intended events. An event is characterized by two elements : the *what* and the *when*. The *what* was relatively simple to explicit. It must be a way to restrict the generating grammars. The *when* needed more research. The first attempt to specify the *when* used a pattern recognition approach. "If I see A generated then I trigger event E". Unfortunately, pattern recognition is hard and heavy. There is a better solution. As every element is generated according to a grammar, at any point it is possible to know what pattern (of the grammar) has been and is being generated.

That is why the grammars of this project have been provided the ability to trigger grammar restricting events. In the following, you will see how the musical model is concretely build, the key points of the realization and how the code can be used to parametrize good generation.

2 Technical aspect

This section presents in detail the scheme of implementation and the key elements with theirs functionalities.

2.1 Model

This project proposes a ruled generation of music based on grammars, enhanced with events abilities.

The melodies are modeled through four different elements representing a sequence of musical cells (figure 1). A cell is defined by one chord, one total duration (called root rythm), one rythmic pattern and a sequence of tones. Each of those four components are generated from user-defined grammars. To avoid confusion between a character object of programmation and a character generated by a grammar, I call "word" the non-epsilon terminals generated by a grammar.

The chord grammar simply generates a sequence of chords, one per cell, ideally a harmonic progression. The root rythm grammar gives the duration of the cell, hence one duration per cell is required. The ryhmic pattern is represented with a sequence of rhythic cells. One word is one rhythmic cell. A rhythmic cell is a sequence of durations that compose a rhythm, for example figure 2. Since the total duration of the rhythmic cell may not match the imposed duration of the cell, the rhythmic cell has to be strectched or shrinked. If the figure 2 is generated for a cell with root duration of ♩, it will be transformed in figure 3

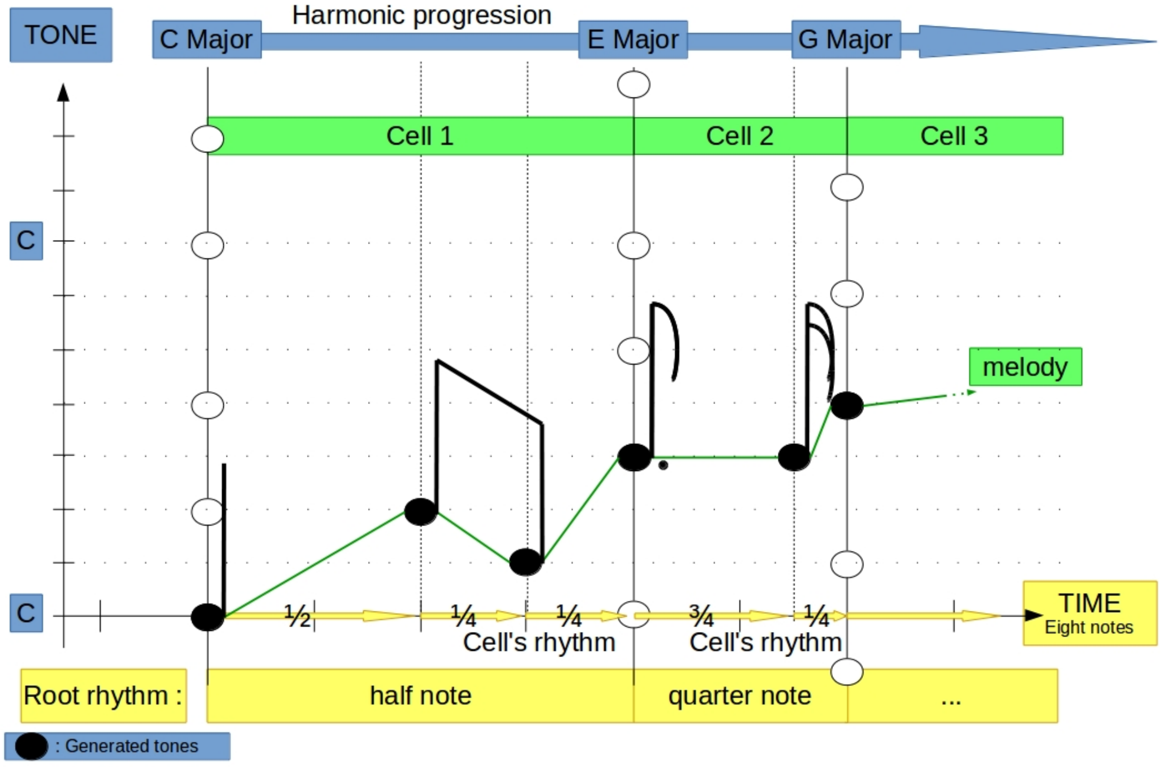


Figure 1: Melody representation

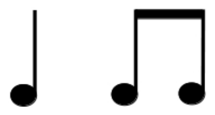


Figure 2: Sample rhythmic cell

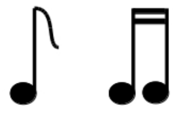


Figure 3: Shrinked rhytmic cell

Finally the sequence of tones determines the height of the notes present in the cell. Unlike rhythmic cells, the melody grammar generates tone-by-tone words. This choice represents the fact that rhythm is mostly repetitive and appears as small units while the melody is a flow of tones. Obviously, the number of tones must match the size of the rhythmic cell (three in our small example).

At last, the first tone of a cell is required to be part of the chord of that cell. A note of any octave may be part of a chord, as long it has the 'right name'. The tone C at any octave is part of the C-Major chord.

To summarize, one cell :

- Starts with a tone of a provided chord
- Lasts according to root rhythm
- Contains a melody based on the rhythmic cell

With the four grammars, one cell is generated as follows :

- Chord grammar generates one more chord c
- Root rhythm grammar generates one more word
- Rhythmic cell grammar generates one more word of size s
- Melody grammar generates s tones such that the first one is in c

The sequence of cells is generated until some ending condition is reached, like the chord grammar reaches its end. If the generation fails, for example the melody grammar did not allow to generate first tone of cell inside the chord, the longest successful sequence of cells is returned. For more details about how multiple possibilities are managed and the specification of closing conditions please refer to section "Generation Algorithm".

2.2 Grammar and Refinements

As stated before, all generations are ruled by grammars. Because the aim is randomized generation with some control, alternatives must be provided with some weight that will indicate which choice to favor (figure 4).

```
A ::= 'c1' A       weight = 1.0
    | 'c2' 'c3' A  weight = 1.0
    | ϵ            weight = 0.5
```

Figure 4: Weighted Grammar

The possibility of writing arbitrary grammars to generate the four melodic components already reaches a good flexibility. But this may not be enough. Indeed, there is very few control and communication possible between the differents components. The construction does not allow statements like “if C-Major is generated, then the melody must be C-E-G”. But such concept is very useful to restrict and control generation.

That’s why grammars are extended with the *refinement* capability. In a nutshell, a refinement is a message sent by a grammar A to a grammar B (A may equal B) that will restrict the generated sequence of B.

So, what implements a refinement ? Nothing else than another grammar. Note that the types of grammars must be equal. To refine the melody grammar, another melody grammar is required. Refining grammar A with A’ consists in taking the conjuction of both grammars. This means that both must agree on generated words. What refinements do is a generalization of the following example.

Recall the two main purposes of a grammar: Parsing and generation. In parsing algorithms, a grammar is used to accept and derive abstract syntax tree of an input. The generation uses grammars to produce a sentence that matches the grammar. In the current case, grammars are used as generators. But what happens if one wants to restrict the generated sentence ? He could put itself between parsing and generation and ask the grammar to parse a regular expression. The grammar rejects the regex if it is not feasible or returns the parsed regex with correct choices where words were not specified.

To get refinements, simply replace the regular expression by another grammar. The refinements are then simply inserted in the emitting grammar and behave like ϵ terminal that send messages to target grammar.

The transformation in grammar notation of the example mentionned above : “if C-Major is generated, then the melody must be C-E-G” is given in figure 5. However, this refinement is very restricive and will likely fail unless initial melody grammar is flexible.

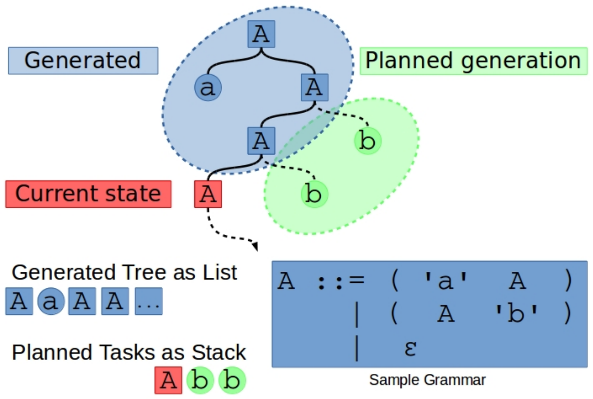
```
// chords grammar
Chords ::= .........
     /* replace all occurences of 'C-Major'
      * terminal by rule CM
      */

// new rule to send refinement
// before generating 'C-Major'
CM ::= RefineMelody(RM) 'C-Major'

// melody refinement
RM ::= 'C' 'E' 'G'

// melody grammar
Melody ::= .........
```

Figure 5: Refinement sample

Therefore this project uses grammars both as generator and parsor. That’s why the term of Parsing Tree is used later, still Generation Tree may have been more accurate.

2.3 Parsing Tree

`ParsingTree` is the class of the program that stores an intermediate state of a grammar (figure 6). An intermediate state of a generative grammar represents which rules and terminals have been generated and what is left to generate. Intuitively, the generation is a path through the grammar’s rules. The intermediate state describes which concrete path has been followed and what is planified to follow, which can contain non-deterministic choices to make. To archieve this, `ParsingTree` stores two lists of grammar elements : The produced sentence and the stack of rules to apply. If the stack is empty, the generation is over.



```
A ::= ( 'a'  A  )
    | (  A  'b' )
    |   ε
```

Figure 6: Intermediate state

`ParsingTree` also maintains a list of received refinements (also instances of `ParsingTree`) that restrict the generated sequence. The nexts possible words are defined by the intersection of the nexts words of the main grammar and all the refinements.

The main functionnality of `ParsingTree` is the function `nexts`. Upon `nexts` call, `ParsingTree` will generate a list of *probable* `ParsingTree` that have generated one more word. The restriction of *probable* is crucial. Recall that alternatives are weighted, they can thus be mapped to probabilities. Since the user-provided grammar may lead to unbounded possibilities (left-recursive grammars), computing the probability of each one is a good way to drop unprobable solutions and bound the result.

2.4 Generation Algorithm

As evoked in section “Model”, this section presents how the generation algorithm manages multiple solutions and what are the possible ending solutions.

Taking everything into account, the generation problem is hard to express. There are four different grammars (Chords[C], Root rhythm[RR], Rhythmic cell[RC] and Melody[M]). For each `nexts` call, a `ParsingTree` may return multiple solutions and for a given cell the M grammar is not independant of C and RC. This means that They cannot be generated independantly and combined afterward. The implemented generation algorithm is approximative and a good backtracking or equivalent approach would give better chances of success. However it is enough with few constraints when the theorical tree of all possible solutions has a large proportion of valid solutions.

There is how one cell is generated (figure 7). The four grammars are represented with `ParsingTree` to store intermediate state. At the first step, there is the four-tuple `C, RR, RC, M`. The call `C.nexts` derives a first bunch of childrens. They are limited according to a fixed bound. The selection of childrens (unlike in the picture) is made randomly, according to probabilities derived from weights of the grammars rules. Then successively repeat this operation with `RR.nexts`, `RC.nexts` and `M.nexts`. When the new cells are generated `C', RR', RC', M'` if one fulfills the closing conditions (see next) then return that solution, otherwise generate another cell.

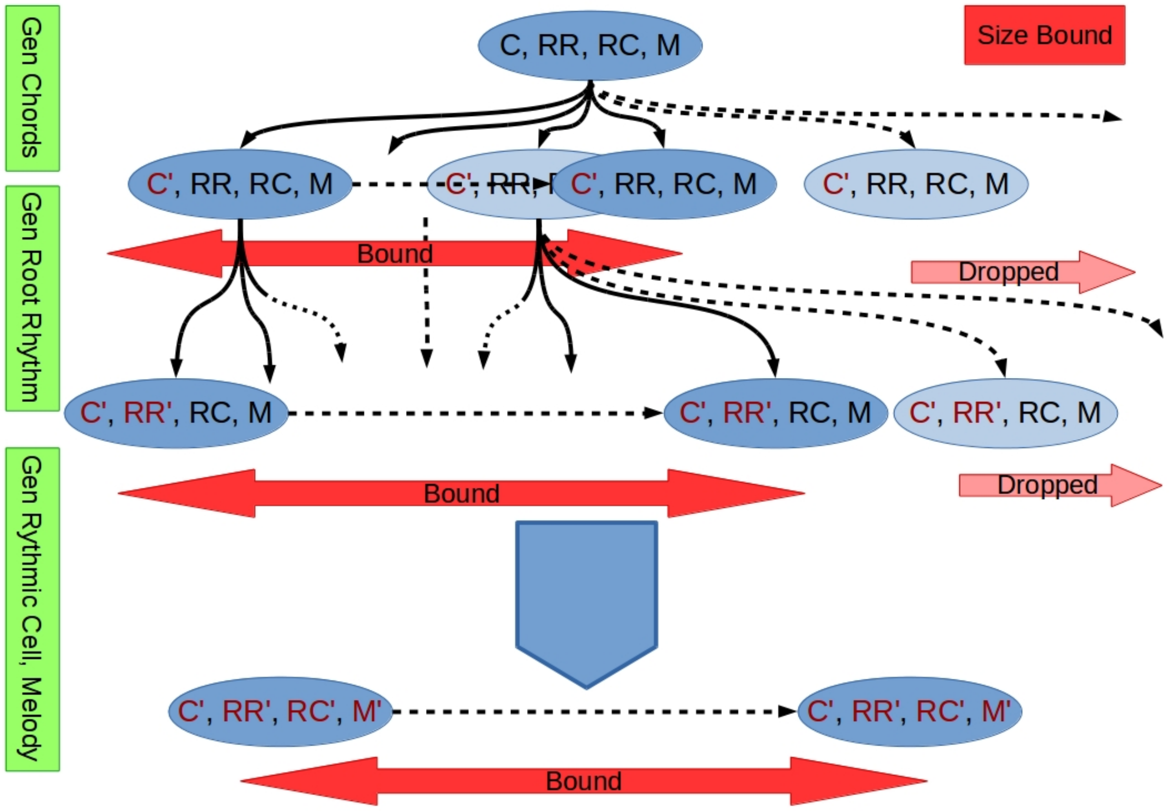


Figure 7: Cell generation

Unlike the generation, the closing conditions are rather simple to formulate. The generation will always try to stop when the chord grammar reaches its end. This makes sence since the harmonic progression is the backbone of the melody. Then for each other grammar, the user can decide if they have to finish (the stack of planned generatio is empty) as well or not. This feature proves useful when one wants to precisely control the end of the generation (figure 8).

```
// generative grammar
A ::= B A | E

// Body of grammar
B ::= ... /* some random generation */

// End of grammar
E ::= ... /* this must occur at end */
```

Figure 8: End Control

3 Composition of an Example

This section presents how to write grammars to generate a melody. At first, very simple grammars produce poor result, then incrementally using more features will enhance the generation to archieve an acceptable output.

To see source code and run samples according to different steps, feel free to clone the git repository of the project and checkout to the "presentation" branch.

```
> git clone https://github.com/vtpittet/ScalaMusicGeneration
> cd ScalaMusicGeneration/MusicInterface/
> git checkout presentation
> sbt
> run
```

Then, you can select the class to run according to the name of the step you want to hear. The source code is located in samples/Presentation.scala

For the sake of readability every grammar is written according to C-Major scale.

3.1 Purely Random

We first start with a simple deterministic chord sequence, the standard Tonic-Subdominant-Dominant-Tonic harmonic progression

```
Chords ::= 'C-Major' 'F-Major' 'G-Major' 'C-Major'
```

Root rhythm and cell ryhthm are infinte sequences of one single element.

```
Root ::= '𝅗𝅥' Root
Cell ::= '♩  ♪♪' Cell
```

Finally the tones, each step generating randomly one tone in the C-Scale in the base octave (figure 9). All weights for alternatives are 1.0.

```
Tones ::= ('C' | 'D' | 'E' | 'F' | 'G' | 'A' | 'B') Tones
```

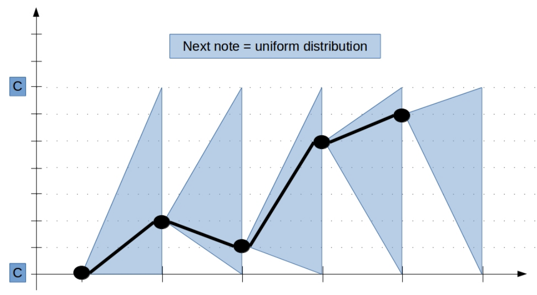


Figure 9: Uniform Random Tones

The generation follows default ending conditions. Only the chords sequence has to finish. Thus there is no trouble with the infinite grammars.

Generated melody is very poor. Let's see how to improve it.

3.2 Flowing Melody

Obviously the first fix is addressed to the tones generation. The idea is to remember the previously generated tone to produce a close one to avoid big steps in the melody.

```
C ::= 'C'
    (  B, w = 2.0
    |  C, w = 1.0
    |  D, w = 2.0)

D ::= 'D'
    (  C, w = 2.0
    |  D, w = 1.0
    |  E, w = 2.0)

// similar construction for E, F, G, A, B
E ::= ...
```

The C-rule produces a 'C' terminal and then can go to rule B, C or D, that behave similarly (figure 10).

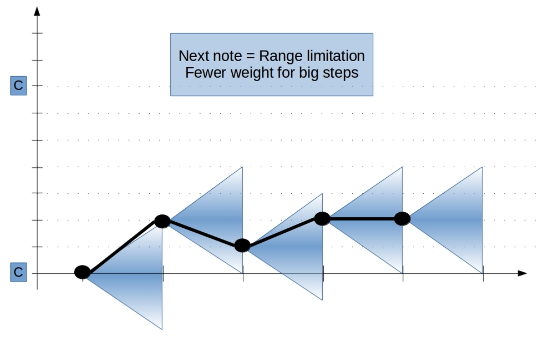


Figure 10: Weights and Bounded Tones

This is already much better. In real Scala code, all rules ran be represented with a single **def** that takes the initial tone as parameter and returns the rule. This functionnal approach allows to describe more complex grammar with only few lines that may have as additionnal parameter

Keeping the information of global movement of melody, the rules can adapt to follow global movement to generate a "flowing" melody (figure 11).

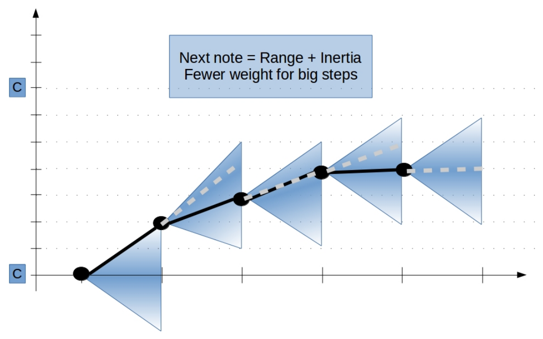


Figure 11: Inertial Tone

3.3 Harmony and Rhythm Variations

Simply using more alternatives for chords and rhythms generation produces more diversity. All weights are set to one. Chords has now three alternatives

```
Chords ::= 'C-Major' B 'C-Major'

B ::= 'G-Major'
    | 'F-Major' 'G-Major'
    | 'F-Major' 'G-Major' 'G-Seventh'
```

Each rhythmic component has two alternatives. This produce four different rhythm for the cells.

```
Root ::= ('♩' | '𝅗𝅥') Root
Cell ::= ('♩  ♫' | '♩.  ♪' Cell
```

This simple way adds diversity and make the melody more interesting.

3.4 End Specification

In all the previous samples, the closing condition was on the chord sequence only. The produced melodies end abruptally and leave a feeling of incompleteness. To solve this problem, the first point is to change the closing condition and require that every generated sequence must end simultaneously. Then the grammars are adapted to be finite and there is only one way to finish the generation

The grammars must follow this pattern :

```
A ::= (Body A, w = 1.0) | (End, w = 0.5)

Body ::= // list complex choices

End ::= // restricted end of generation
```

The bodies of grammars are unchanged and the ends are simply 'C' for Tones (we want to end on the tonic) and '𝅗𝅥' for Root and Cells to close with a long single note.

Clearly the music ends much more neatly. However some runs still have a bad end. When the second to last note is very distant to the required last note (one octave lower for example) the end appears out of nowhere. The final enhancement fixes this problem using refinements.

3.5 Refinements

The proposition is to anticipate the end of the music and add a constraint to the Tones grammar that will make it converge towards the ending tone over the generation of n tones.

This converging grammar is simply represented by a sequence of n rules. The first one can generate n different notes over and below C, the second one $n-1$, ... the nth one must generate the intended 'C' (figure 12).

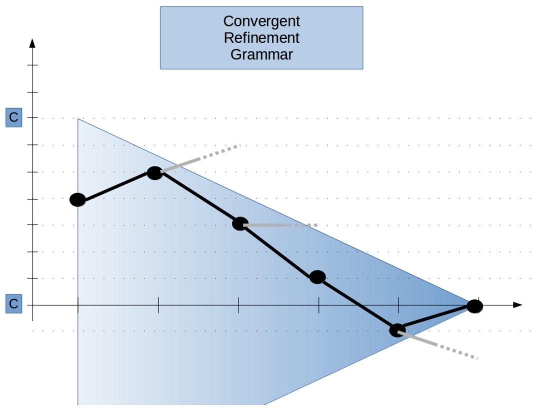


Figure 12: Convergent Refinement

Finally the refinement is introduced inside the sequence of chords.

```
// three times a harmonic sequence of chords
// then send refinement and produce last sequence of chords
FinalChords ::=
  Chords Chords Chords RefineMelody(RM) Chords

// converging grammar for melody
RM ::= ...
```

4 Conclusion

4.1 Qualitative Evaluation

Weakness of harmonic constraint It may appear that restricting only the first tone of a cell to match the provided chord is too weak. Indeed, it does not produce a complete harmonic progression on each note and the melody has a lot of freedom.

However, this constraint is not useless. People were asked to compare generations with and without this restriction and gave an interesting observation. The melodies themselves were hard to compare, but in the latter case, the rhythm seemed degenerated. One interpretation is that the presence of the harmonic constraint at the beginning of the cells is passively perceived and produce an emphasis that evokes a reference point to interpret the music.

On the other hand it is still possible but more cumbersome to generate a melody where all notes match a harmonic progression. Use the root rhythm as the only rhythmic component and produce one infinite sequence of singleton rhythmic cell (figure 13)

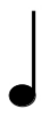


Figure 13: Singleton Rhythmic Cell

Generation algorithm The generation algorithm is simple and not very efficient. It cannot generate when the rules are too complex. Additionnaly, the usage of probabilities to limit the result, a good solution for simple cases, make it hard to predict the result. Suppose one wants to control the end of the melody and adds the tone grammar to ending conditions. If the ending path is long and its weight is small, it will never be kept long enough to reach the ending state. Thus when the chord grammar will reach its end, there will be no candidates in melody that ends as well ant the algorithm will nearly always fail.

Success for quick and simple runs There are limitations, but if we manage to respect them, the result is good. The main limits are in the generation algorithm and the model itself proves to work. The initial wish to avoid heavy cases analysis, situation recognition and replace them by generation knowledge appears to be a valid approach. It would be interesting to see now how one, more constraint-based is related to the other one, generation-based.

4.2 Further Improvements

To close this report, I suggest a list of further developments and improvements that could be brought to the project.

Deuniformize refinements and main grammars gestion For the sake of generality and simplicity, refinements and main grammars are implemented exaclty the same way. This means that a refinement grammar could send a refinement message. Removing this ability will allow to improve the chances of success of the algorithm.

More formal specification of structures The overall generation structure is hard to represent. Roughly, there are search tree whose nodes are search tree themselves. Maybe this can be simplified into a whole tree, simpler to manage. Or a good specification could lead to a unique interface, clarifying the code and removing duplication.

Output parsing tree structure `ParsingTree` stores every step taken by its represented grammar. In other words, it stores the prefix traversal order of the generated abstract sytax tree. This information is easily collectable and would give an interesting view of the generated structure.

Integration with Scala Music Generation project This project was build on top of abstraction provided by the initial Scala Music Generation project. The presented samples do not use full power of the underlying code. Eventually the goal is to generate small random cells using the current project, combine and manipulate them to build structured pieces with repetitions, variations and modulations.