


Musikla: Language for Generating Musical Events

Pedro M. Silva 

Dummy University Computing Laboratory, Portugal

My second affiliation, Country

<http://www.myhomepage.edu>

johnqpublic@dummyuni.org

José João Almeida 

Algoritmi, Departamento de Informática, Universidade do Minho, Braga, Portugal

jj@di.uminho.pt

Abstract

In this paper, we'll discuss a simple approach to integrating musical events, such as notes or chords, into a programming language. This means treating music sequences as a first class citizen. It will be possible to save those sequences into variables or play them right away, pass them into functions or apply operators on them (like transposing or repeating the sequence). Furthermore, instead of just allowing static sequences to be generated, we'll integrate a music keyboard system that easily allows the user to bind keys (or other kinds of events) to expressions. Finally, it is important to provide the user with multiple and extensible ways of outputting their music, such as synthesizing it into a file or directly into the speakers, or writing a MIDI or music sheet file. We'll structure this paper first with an analysis of the problem and its particular requirements. Then we will discuss the solution we developed to meet those requirements. Finally we'll analyze the result and discuss possible alternative routes we could've taken.

2012 ACM Subject Classification Computing methodologies → Language resources

Keywords and phrases Umbundu, Angola Languages, Morphological Analysis, Spell Checking

Digital Object Identifier 10.4230/OASICS.SLATE.2020.23

Funding *Pedro M. Silva*: (Optional) author-specific funding acknowledgements

1 Introduction

Musikla stands for Music and Keyboard Language. Our goal is to develop a DSL (*Domain Specific Language*) that allows treating musical events as regular data in a programming language. More than generating these musical events offline, we want to be able to easily declare keyboards that map keys to expressions that either mutate the state or play musical events (or even both).

The project can be partitioned in three different, modular layers: inputs, the language, and outputs. While music events can be described as code literals inside our language, they can also originate from many other sources (such as files or physical devices such as pianos). After being processed by our language, they are then emitted as a stream of musical events to the **Player** component, which then multiplexes those events into however many outputs the user defined.

While the development of both the input and output layers, as well as their many respective components, presents by itself many interesting challenges that could be discussed, we will instead focus this paper on the aspects of the middle layer: the *interpreter*, while acknowledging the existence (and their effects) of the layers that wrap around it.

As such, the problem of developing the interpreter can be divided into two parts: the syntax used for describing the notes and the operators that compose them inside the language; and the semantics of the generated events, how they are stored in memory, and



© Pedro Miguel Silva, José João Almeida;
licensed under Creative Commons License CC-BY

42nd Conference on Very Important Topics (SLATE 2020).

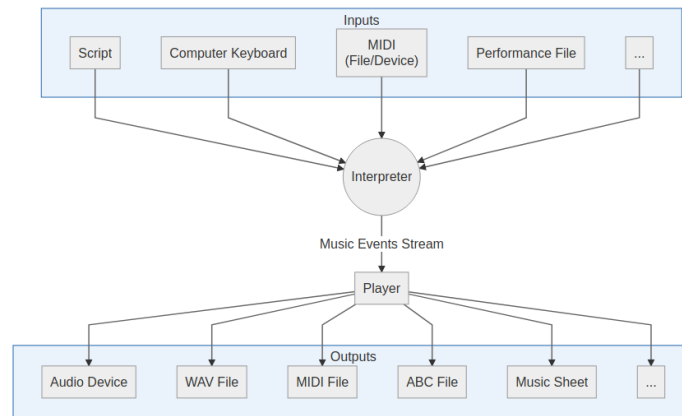
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:11

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

23:2 Musikla: Language for Generating Musical Events



■ **Figure 1** The three main layers of the project.

45 how their temporal properties (start time and duration) are handled without forcing the
46 programmer/user to always manually type them.

47 Designing the syntax for describing those musical expressions, especially given our strong
48 desire to make those musical expressions first class citizens like other primitive data types in
49 most programming languages (such as numbers, strings or arrays are), did unearth some
50 challenges. To minimize the learning curve for new users, and avoid reinventing the wheel,
51 we decided to adopt a subset of the very popular note declaration syntax from the ABC
52 Notation project[4, 6] and integrate it with our language.

53 As for the execution model, we decided to go with a tree-walker interpreter[8]. Although
54 computationally slower than other alternatives (such as a bytecode virtual machine), the ease
55 of implementation allowed us to prototype and develop features extremely fast. And with a
56 more mature and stable language in the future, there is always the potential to rewrite the
57 interpreter if performance or latency ever reveal themselves as potential problems.

58 The simplest way of generating such musical events in a programming language is to use
59 already common, *low-level*, programming mechanisms, such as using a procedural approach
60 where the user creates each event manually by calling a function and providing as parameters
61 all the events' information, such as it's timestamp and duration. This is the approach used
62 by some of the existing languages in this space, such as *SonicPi*[5].

■ **Listing 1** Example of a hypothetical imperative API for creating events

```
63  
64 play_note( 0, 100, 'A' );  
65 play_note( 100, 50, 'B' );  
66 play_note( 150, 200, 'C' );  
67
```

68 Instead we decided to follow a more *functional* approach, with custom syntax and
69 operators, as well as the hability to describe those events in a single expression. Musical
70 events are treated as sequences, and as such can be stored in variables, passed around inside
71 functions and trasformed. So, for musical events, we will be exploring a way to define them
72 in code, as *musical literals*.

■ **Listing 2** Our proposed declarative syntax that calculates timings implicitly

```
73  
74 play( A B/2 C2 );  
75
```

2 The Problem and its Requirements

There are two important requirements we need to consider when evaluating possible solutions to this problem: the ability to produce music interactively, and to produce music lazily.

The first requirement, **Interactivity**, relates to our goal of not only being able to generate music offline, but also in a live environment: give the user the ability to program several snippets of musical events, and then control them through a virtual keyboard or through other interactive means.

The second requirement, **Laziness**, refers to a concept that is familiar in functional programming languages: values are generated when we need them, not earlier. In our case, this implies that a musical sequence could be potentially infinite (like an infinite repetition of some arrangement). If playing this music live, the musician could determine to stop this arrangement sooner or later.

Given these two requirements, we can conclude we **cannot** generate all music events at the start and then sort them to play them in order. Because of that, the events must always be sorted already.

► **Lemma 1 (Total Order).** *All operators must return a sequence of events that respects our time unit's total order.*

Goals

We can then summarize our main goals for this language as follows:

- **Declarative** Music sequences are described in a declarative (rather than imperative) fashion.
- **Dynamic** Introduce programming or mathematical concepts, like functions and variables, to the music world.
- **Interactive** Make it possible to create interactive keyboards out-of-the-box that integrate with all features provided by the language.
- **Lazyness** Make lazyness for musical sequences the default, generating only events as they are need.
- **Rich events** Music sequences can describe complex musical arrangements, containing simple notes, rest, chords, voices, and more.
- **Multiple Inputs** Besides allowing musical arrangements to be declared inside our language, also allow for them to be imported and converted from multiple sources, like MIDI files and devices.
- **Multiple Outputs** Store or write to multiple different outputs the music events generated inside our language.
- **Extensibility** Make it easy to extend and customize the project, without needing to fork or recompile or hack it's internals.

Data Model

The basic premise is that expressions can generate a special data type: **Music**. Music is simply a sequence (or stream) of ordered musical events.

A musical **Event** can be one of many things, such as a *note*, a *chord*, or even more implementation-specific events like MIDI messages[7]. While all events must have a start time, some events can be instantaneous (events with a duration of zero time units).

The time unit used does not need to be a common time measure, like seconds or milliseconds, and can be really anything so long as it has a **total order**.

23:4 Musikla: Language for Generating Musical Events

120 Operators

121 Operators are special operations defined at the syntactic level that allow *music* to be composed
122 in different ways, such as concatenated, parallelized or repeated. Many of these operators can
123 have equivalent functions available through the language that provide more costumization
124 (such as a parallel function that stops when the smaallest operand stops, instead of the
125 longest).

```
126 Concatenation Music1 Music2 ... MusicN  
127     type List[Music] -> Music  
128 Parallel Music1 | Music2 | ... | MusicN  
129     type List[Music] -> Music  
130 Repetition Music * Integer  
131     type Music, Integer -> Music  
132 Arpeggio Chord * Music  
133     type Chord, Music -> Music  
134 Transpose Music + Integer and Music - Integer  
135     type Music, Integer -> Music
```

136 It is also useful to establish that while most operators work on sequences of musical
137 events, they can also accept a singular event as their argument: one event can be trivially
138 converted into a sequence of one element. Such ocorrence is so common and trivial that the
139 conversion should therefore be implicit whenever necessary.

140 Grids

141 Another type available in our language are grids. Also known in most music applications
142 as the process of quantization [2]. The reason it is so useful in our language is that when
143 receiving input as musical events from a live keyboard, their timings are naturally more
144 prone to having small discrepancies that can become more apparent when we then mix them
145 with generated musical events (which have precise timings).

146 Having events always aligned with such a grid can also make computations and trans-
147 formations of such events easier and simpler, which is always a plus for our language.

```
148 Create a grid Grid(size)  
149     type Fraction -> Grid  
150 Aligning grid::align(music)  
151     type Grid, Music -> Music  
152 Compose Grids Grid::compose(grid1, grid2, ..., gridN)  
153     type List[Grid] -> Grid
```

154 We can see that apart from the basic operations of creating a grid and aligning events to
155 said grid, we also want the ability to compose multiple grids (of different precisions). We
156 will approach this matter in more detail later.

157 Keyboards

158 A core part of the language is our hability to declare keyboards, which we can describe as
159 mappings between *Keys* and *Musical Expressions*.

160 Each expression can mutate the state (changing variables or calling functions), return
161 some music (sequence of musical events) to be played, or both.

162 Some of the operations we want to be able to perform with keyboards are as follows:

```

163 Create a keyboard keyboards\create()
164     type () -> Keyboard
165 Binding a Key keyboard::register(key, expression)
166     type Keyboard, Key, Expression -> Keyboard
167 Mapping a keyboard keyboard::map(transformer)
168     type Keyboard, ( Music -> Music ) -> Keyboard
169 Aligning with a Grid keyboard::with_grid(grid)
170     type Keyboard, Grid -> Keyboard

```

171 3 Implementation

172 The reference implementation for this system is written in Python, although the approach
 173 here should be language agnostic.

174 One of the features that Python boasts (but are certainly not exclusive to it) that have
 175 eased our implementation of the language are generators[3]. They integrate very nicely into
 176 both our concept of emitting musical events as sequences (or iterators, as they are called
 177 in Python and other languages), as well as into our concept of laziness, where events are
 178 generated on demand when needed, and thus infinite musical sequences can be handled easily.

179 Context State

180 To keep track of the *cursor* (the current timestamp where the next event should start) each
 181 operator in our language is implemented as a function call that receives an implicit **Context**
 182 object. While here we'll mostly focus just on the methods related to time management
 183 provided by the context, it can be used to store other types of information, like the default
 184 length of a musical note, for instance, to avoid forcing the user to type it out all the time, or
 185 the tempo at which it is to be played.

186 It is important to keep in mind that there might be more than one context in execution
 187 at the same time. This can be most obvious with the use of the parallel operator, where
 188 each operand must run concurrently (and thus could not share the same context).

189 Let's describe what kinds of functionality our context should provide.

```

190 cursor(ctx) Return the current cursor position
191 seek(ctx, time) Advance the cursor to the given position
192 fork(ctx) Clone the parent context and return the new one. Allows multiple concurrent
193     contexts to be used
194 join(parent, child) If the child's cursor is ahead, make the parent context catch up

```

195 3.1 Operators

196 Basic Events

197 The basic building block of our system are the **Note**, **Chord** and **Rest** events. We can
 198 use the current *context* to determine the event's timestamp, as well as its default duration
 199 (in case the user does not explicitly state one). Any event(s) that is/are not captured in a
 200 variable or passed to a function are implicitly played.

201 **Listing 3** Creating a Note Event

```

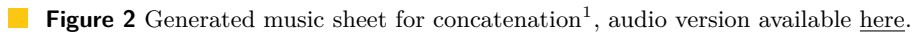
202 c ' 1/4
203

```

204 **Concatenation**

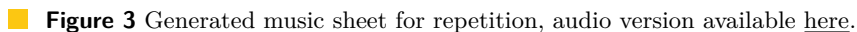
■ **Listing 4** Snippet of the song *Wet Hands* by C418

S4/4 T74 L/8 V90;
A, E A B ^c B A E D ^F ^c e ^c A3;



■ Listing 5 Intro to Westworld's Theme by Ramin Djawadi

I1 S6/8 T140 L/8 V90;
A*11 G F*12



¹ Rendered with \$ABC\$ UI. Some hand made changes made for clarity.

■ **Listing 6** Snippet of the song *Soft to Be Strong* by Marina

```

229 T120 V70 L1;
230 r/4 ^g/4 ^g/4 ^g/4 ^f/2 e/8 ^d3/8 ^c2 | [^Cm] [BM] [AM] [BM]
231
232

```



■ **Figure 4** Generated music sheet for parallel, audio version available [here](#).

The merge sorted function receives N operands and creates a buffer with the size N . For each operand it *forks* the context, so that they can execute concurrently and each will mutate their own context only. It then requests one single event for each operand.

After the buffer is prefilled (meaning it has at least one event for all non-empty operands), the algorithm finds the earliest event stored in the it. Let's assume it is stored in the K index of the buffer, with $K < N$. The method emits the value stored in `buffer[K]` and then fills requests the next event from the K operand (storing `null` if the operand has no more events to emit). It then repeats this step until all operands have been drained.

3.2 Integration in a Programming Environment

Apart from generating musical events from somewhat static instructions, our goal is to have those events integrate into a programming language in the same way integers, floats, strings and booleans do: as data that can be stored, passed around and manipulated. This, of course, must still retain all the properties we've laid out for our sequences of events: being lazy and always being ordered.

Variables and Functions

All expressions that are assigned to a variable run in a forked context, with its cursor set to zero initially. Musical expressions inside variables are still lazy (meaning they only calculate each musical event when the variable is first used, not declared) but the events are cached to prevent the calculations from being performed every time. This cache is then garbage collected when the variable is no longer in use.

Since events declared inside variables have their start time set relative to zero, it needs to be calculated each time the variable is used to replace it with the correct value. This highlights an important aspect of our language: the need for each musical event to be immutable, and any changes made to them to actually be implemented as new instances of the event.

This works well enough because those events are very lightweight objects, and the benefits of not having their values mysteriously changed midway during execution outweigh the

23:8 Musikla: Language for Generating Musical Events

260 small cost of a possible unnecessary allocation of an event that would only be used in one
261 place instead of many.

262 Function calls, on the other hand, pass the current context to the inside of the function,
263 so that any events played there now their correct times.

264 When integrating functions into our language, we decided to keep the semantics simple.
265 Emitting musical events inside a function is similar to its return value being an iterator that
266 gives out the emitted events on demand. This means that a value cannot both emit musical
267 events, while also returning other values manually through a return statement.

268 There is no syntactic marker to distinguish regular functions from musical-emitting
269 ones. Instead, the language runtime starts executing each function as a regular one, and
270 automatically switches its execution mode into a generator-like implementation once the first
271 event is emitted. Any return statements that are evaluated after this point must have no
272 value (thus preserving the feature to early-stop a function). If they do try to return a custom
273 value, a runtime exception is triggered.

274 Here we can see a small snippet of the beginning of *Fugue 2 in C minor in Book I of*
275 *the J.S. Bach's Well-Tempered Clavier*, and how using functions and variables can help us
276 visualize the structure behind music.

■ **Listing 7** Example of repeating the same note

```
277 fun fugue ( $subj, $resp ) =>  
278   ( $subj $resp | stretch( r, $subj ) ( $subj + 7 ) );  
279  
280  
281 S8/4 T140 L/4 V120;  
282  
283 $subj = r c/2 B/2 c G _A c/2 B/2 c d  
284   G c/2 B/2 c d F/2 G/2 _A2 G/2 F/2;  
285  
286 $resp = E/2 c/2 B/2 A/2   G/2 F/2 _E/2 D/2   C _e d c  
287   _B A _B c   ^F G _A F;  
288  
289 play( fugue( $subj, $resp ) );  
290
```



■ **Figure 5** Generated music sheet for fugue example, audio version available [here](#).

Grids

To define a grid there is only one parameter required: the length of it's cells. When aligning musical events, anything that falls inside each cell will be pushed to the closest edge of the cell.

Grids are highly customizable too, however. They have multiple parameters, such as **forgiveness** and **range**, that determine when an event is affected by the grid (depending on how close it's start time is to the edge of the cell). Each parameter can even be customized separately for the left and right sides of the cell's edge.

Let's take a look at an example of a grid. In this example the grid has a cell size of **1**. We define the same values for both left and right sides just for the sake of this demonstration, but each side could have different values.

Listing 8 Declaring a grid

```
$grid = Grid( 1,
  forgiveness_left = 125, # 1/8
  range_left = 375, # 3/8
  forgiveness_right = 125, #1/8
  range_right = 375 # 3/8
);
```

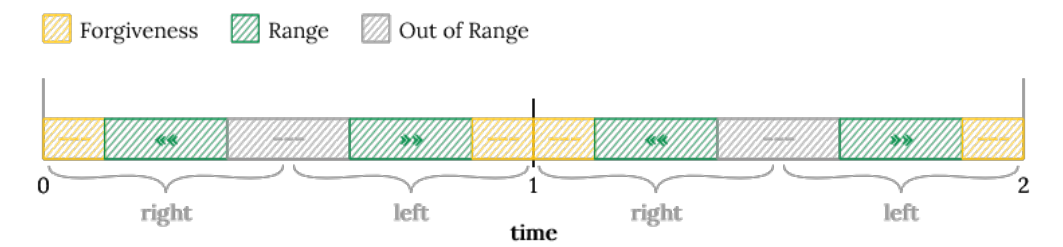


Figure 6 Representation of two cells from this grid.

We can see in this timeline two cells (each with a size of 1). Any events that fall in the yellow and grey areas are ignored (meaning their timestamps are not changed) while events in the green areas are pushed to whatever edge is closest. But even this behavior can be customized, forcing events to always go to the previous edge cell, or always to the next.

It is then trivial to see how we could compose multiple grids in sequence, each with different ranges (green areas) that capture different events and align them accordingly.

Keyboards

Finally we can combine all the systems we've described above, from musical expressions, grids, variables and functions, and devise a compact way of describing virtual keyboards.

To make the process of designing keyboards less verbose, we've added syntactic sugar to this process, that is translated in the background to regular function calls registering each key binding.

While a picture maybe worth a thousand words, a good example is worth maybe even more. So here we can take a brief look at the workflow for defining two keyboards (that are active at the same time). The first keyboard has all the musical keys (the chords and single notes we want), all aligned by a custom grid.

23:10 Musikla: Language for Generating Musical Events

The second keyboard binds to the up and down arrow keys and allow us to change the virtual instrument through which we play the sounds of the notes in the keyboard (those instruments can be identified by an integer and usually follow the General MIDI standard[1]).

■ **Listing 9** Creating a keyboard that can play multiple instruments

```
$inst = 0;

fun spin_instrument ( ref $instrument, $change ) {
    $instrument = $instrument + $change;

    setinstrument( $instrument );
};

@keyboard {
    a: [^Cm];    s: [BM];    d: [AM];    f: [EM];    g: [^Fm];
    1: ^c;       2: ^d;       3: e;        4: ^f;       5: ^g;
    6: b;        7: ^c';      8: ^d';      9: e';
}::with_grid( Grid( 1 / 16 ) );

@keyboard {
    up: spin_instrument( $inst, 1 );
    down: spin_instrument( $inst, -1 );
};
```

Keyboards are objects (that we could save in a variable for example) and that can perform many operations, like unions and intersections, or maps and filters. They can be enabled and disabled at runtime, and their keys can be simulated to be pressed and released.

More than that, we don't need to restrict ourselves to computer keyboards. We can for instance, define bindings between MIDI events and musical expressions, so that when we connect a piano keyboard to our computer, we can use each piano key to play more than a single note.

Since like we've seen keyboard keys are not limited to computer keyboards, we can imagine the possibilities of events we could listen to: knobs, mouse buttons, the mouse scroll wheel. We could even create an event that could, for example, listen on a socket and trigger when a message is received, allowing in that way our musical applications to be controlled remotely.

The result is that our keyboards are extremely extensible and allow for a great deal of creativity. And thanks to our tight integration with the Python language, those extensions can be easily integrated and don't require hacking the source code or recompiling the application.

4 Results Discussion

The scope of this project could have been massive, mainly because implementing a way to fully describe hundreds of years of cumulative musical notation would be a gigantic task. We chose instead to build a solid foundation, always with a strong focus on extensibility.

This hability to extend the functionality of our project, thanks to our easy integration with regular Python code, means that new types of musical events, new inputs or outputs can be added with minimal effort by everyone using our application.

Technically, we also took decisions on how to approach many issues, but it doesn't mean there were no alternatives. For example, to solve our problem of keeping track of the timing implicitly for each event created, we decided to pass around a context variable. There were other possible solutions, like keeping this data in some sort of global variable. Our approach

375 does give us some advantages, such as being able to have multiple contexts in play at the
 376 same time. It does have drawbacks, too. Every function defined in our language must receive
 377 this context to be able to create events at the appropriate time. However, functions defined
 378 in Python do not expect this parameter. Therefore, special conversions must be made when
 379 exchanging functions between both languages.

380 Also, our solution to have variables just offset the timings of each event they contain every
 381 time those variables are used simplifies the process of integrating variables into our existing
 382 semantics of music generation. This solution, however, does not answer other questions
 383 unrelated to the timing, such as: should events stored in variables use the musical instrument
 384 set when they were declared, or when the variable was used?

385 When it comes to keyboards, there are many more possibilities to explore too. While
 386 we've included many examples of working with key presses, both from computer keyboards as
 387 well as pianos (through a MIDI connection), more rich events can be used. Since each event
 388 can also carry parameters with it, we can do more than boolean press/release types of events:
 389 we can model knobs or scroll wheels or other kinds of spatial events into our keyboards.

390 However, while not implementing every excruciating detail needed to match every need
 391 that could ever arise, our work already provides a solid foundation for a musical *DSL* that
 392 while dynamic (with variables, functions and control structures) integrates very well with
 393 established musical standards such as the MIDI protocol and others. And it does this while
 394 also being easily extensible to allow anyone to customize it to its needs.

395 — References —

- 396 1 Gm 1 sound set. URL: [https://www.midi.org/specifications-old/item/](https://www.midi.org/specifications-old/item/gm-level-1-sound-set)
 397 [gm-level-1-sound-set](https://www.midi.org/specifications-old/item/gm-level-1-sound-set).
- 398 2 Quantization music. URL: [https://en.wikipedia.org/wiki/Quantization_\(music\)](https://en.wikipedia.org/wiki/Quantization_(music)).
- 399 3 Pep 255 – simple generators, May 2001. URL: [https://www.python.org/dev/peps/](https://www.python.org/dev/peps/pep-0255/)
 400 [pep-0255/](https://www.python.org/dev/peps/pep-0255/).
- 401 4 The abc music standard - the tune body, December 2011. URL: [http://abcnotation.com/](http://abcnotation.com/wiki/abc:standard:v2.1#the_tune_body)
 402 [wiki/abc:standard:v2.1#the_tune_body](http://abcnotation.com/wiki/abc:standard:v2.1#the_tune_body).
- 403 5 Samuel Aaron, Dominic A. Orchard, and Alan F. Blackwell. Temporal semantics for a live
 404 coding language. In *FARM '14*, 2014.
- 405 6 Guido Gonzato. *Making Music with ABC 2*. December 2019.
- 406 7 Gareth Loy. Musicians make a standard: the midi phenomenon. *Computer Music Journal*,
 407 9:8–26, 1985.
- 408 8 Bob Nystrom. *Crafting Interpreters*. 2020.