

# Semantics of Implicitly Timed Musical Events


Pedro M. Silva 

Dummy University Computing Laboratory, Portugal

My second affiliation, Country

<http://www.myhomepage.edu>

[johnqpublic@dummyuni.org](mailto:johnqpublic@dummyuni.org)

José João Almeida 

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

[jj@di.uminho.pt](mailto: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. First we'll analyze 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** This research was partially funded by Portuguese National funds (PIDDAC), through the FCT – Fundação para a Ciência e Tecnologia and FCT/MCTES under the scope of the projects UIDB/05549/2020 and UIDB/00319/2020. Bernardo Sacanene acknowledges from the Angolan government his PhD grant, through INAGBE (Instituto Nacional de Gestão de Bolsas de Estudos).  
*Pedro M. Silva:* (Optional) author-specific funding acknowledgements

## 1 Introduction

Programming languages are sometimes described as data plus code. However, it is a known fact there is some overlap between the two. While a big chunk of data most programs consume is ingested through some sort of *IO* operation (like reading from a keyboard, a file or a socket), and another chunk is generated dynamically by algorithms at runtime, there is a third chunk: smaller in amount but maybe just as important: constants (or literals) inserted into the code by the programmer.

Virtually every popular language nowadays has custom syntax to allow the programmer to describe some very common (and primitive) data types, such as numbers, booleans or strings. Many modern languages even have syntax for more advanced data structures, such as arrays and dictionaries (or hash tables).

When we cross into the territory of DSLs (*Domain Specific Languages*), there is a multitude of custom syntax for the most varied data types. In this paper we will discuss a proposed solution to the problem of describing musical notes, and more important, musical arrangements and melodies as data values that can be integrated into a programming language.

Such problem can be divided into two parts: the syntax used for describing the notes and the operators that compose them; and the semantics of the generated events, how they are stored in memory, and how their temporal properties are handled without forcing the programmer/user to manually type them. The former is a relatively easy problem, and to minimize the learning curve for new users, we adopted the very popular note declaration syntax from the ABC Notation project, with some minor changes. The latter is the one we'll



© Alberto Simões, Bernardo Sacanene, Álvaro Iriarte, José João Almeida and Joaquim Macedo; 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. 23:1–23:5

OpenAccess Series in Informatics



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

## 23:2 Semantics of Implicitly Timed Musical Events

be discussing in greater detail.

The most straightforward way for generating such timed events is to let the user control the timing manually: setting both properties (start and duration) explicitly for each event, and maybe doing so with regular function calls, thus avoiding the need for extra syntax even. This is the approach used by some of the existing languages in this space, such as *SonicPi*. But this approach is also cumbersome and prone to mistakes though, as one could guess from the following usecase: if the user wants to change one of the earlier events' timing, he would have to manually update the timings of all of the following events.

Instead, we'll take a look at an alternative approach based on custom operators and syntax, each with their own semantic and syntax that enable an expressive way to define those events. We can look at this the same way languages implement other primitive values: most often we can declare static strings, not by manually setting each character in memory, but by writing the string wrapped in special characters. For musical events, we will be doing the same, exploring a way to define them in code, as *musical literals*.

But more than just being able to define those events, we will also be interested in exploring how well they integrate with existing and common programming language constructs, like variables, functions, loops and other control structures.

### 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 when 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 play them in order.

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

### Data Model

The basic premise is that expressions can generate a special data type: **Music**. Music is simply a sequence 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. While all events must have a start time, some events can be instantaneous (events with a duration of zero).

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**.

## 86 Operators

87 Operators are special operations defined at the syntactic level that allow *music* to be composed  
88 in different ways, such as concatenated, parallelized or repeated.

89 **Concatenation** *Music1 Music2 ... MusicN*

90 **Parallel** *Music1 | Music2 | ... | MusicN*

91 **Repetition** *Music \* Integer*

92 It is also useful to establish that while most operators work on sequences of musical  
93 events, they can also accept a singular event as their argument: one event can be trivially  
94 converted into a sequence of one element. Such occurrence is so common and trivial that the  
95 conversion should therefore be implicit.

## 96 3 Implementation

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

99 One of the features that Python boasts (but are certainly not exclusive to it) that have  
100 eased our implementation are generators. They integrate very nicely into both our concept  
101 of emitting musical events, as well as our concept of laziness where events that are not used  
102 are not generated either.

## 103 Context State

104 To keep track of the *cursor* (the current timestamp where the next event should start) each  
105 operator in our language is implemented as a function call that receives an implicit **Context**  
106 object. While here we'll mostly focus just on the methods related to time management  
107 provided by the context, it can be used to store other types of information, like the default  
108 length of a musical note, to avoid forcing the user to type it out all the time.

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

110 **cursor(ctx)** Return the current cursor position

111 **seek(ctx, time)** Advance the cursor to the given position

112 **fork(ctx)** Clone the parent context and return the new one. Allows multiple concurrent  
113 contexts to be used

114 **join(parent, child)** If the child's cursor is ahead, make the parent context catch up

## 115 3.1 Operators

### 116 Note Events

117 The basic building block of our system is going to be a **Note Event**. We can then add even  
118 more events, like chords and rests, while the type of the event might change, the semantics  
119 are equivalent.

120 The function that emits the event receives the current context as an argument, and is  
121 then responsible for creating the event, with its timestamp matching the context's cursor.  
122 After the event is created, it also must move the context's cursor forward through the **seek**  
123 method.

## 23:4 Semantics of Implicitly Timed Musical Events

### ■ Listing 1 Creating a Note Event

```
124
125 function note (ctx) {
126     event = create_note(cursor(ctx));
127
128     seek(ctx, event.duration);
129
130     yield event;
131 }
132
```

### 133 Concatenation

134 We've seen how single events' creation is handled. Now it is important for us to see how  
135 we can combine those events together. And probably the most straightforward operator  
136 of all, concatenation, it simply consumes each event. Each event, as we've seen before, is  
137 responsible for seeking the context depending on the event's duration.

### ■ Listing 2 Algorithm to concatenate musical events

```
138
139 function concatenate (ctx, ...operands) {
140     for (music in operands) {
141         for (event in music(ctx)) {
142             yield event;
143         }
144     }
145 }
146
```

### 147 Parallel

148 The parallel operator enables playing music simultaneously. However our events are emitted  
149 as a single sequence of ordered events. The operator assumes that each of its operands  
150 already follows the convention of emitting a ordered sequence of events. With this in mind,  
151 relies on a *merge sorted* algorithm.

### ■ Listing 3 Algorithm to merge parallel musical events

```
152
153 function parallel (ctx, ...operands) {
154     for (child_ctx, event in merge_sorted(ctx, ...operands)) {
155         join(ctx, child_ctx);
156
157         yield event;
158     }
159 }
160
```

161 The merge sorted function creates a buffer with the same size as the amount of operands  
162 given. For each operand it forks the context so that they can execute concurrently. It then  
163 requests one single event for each operand.

164 After the buffer is prefilled, the algorithm emits the earliest event stored in the buffer,  
165 and requests the next event from that same operand. It then repeats this step until all  
166 operands have been drained.

### 167 Repetition

168 The repetition operand is in a way very similar to the concatenation operator. It makes sense,  
169 since repeating any kind of music pattern  $N$  times could be thought as a particular case of  
170 as concatenation where there are  $N$  operands, all representing the same musical pattern.

**Listing 4** Algorithm for repetition

```
171 function repeat (ctx, music, count) {  
172     for (i = 0; i < count; i++) {  
173         for (event in music(ctx)) {  
174             yield event;  
175         }  
176     }  
177 }  
178 }  
179 }
```

**3.2 Integration in a Programming Environment**

Apart from generating musical events from 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, while still retaining all the properties we've laid out for our sequences of events: being lazy and always being ordered.

**Variables****Functions****4 Results Discussion**

As previously referred, this is the kick-off for a project on the creation of tools and resources for the Angolan indigenous languages. Although we intend to collaborate with institutions and researchers from Angola, this first proof of concept was developed to understand these languages characteristics.

The current morphological analyzer coverage is quite limited. When processing a corpus with XXX different forms, only YYY forms are recognized as derivatives generated by the morphological rules. The other words are recognized either because they are lemmas, or because our current derivation includes word forms directly in the dictionary (instead of proper derivation rules).

While in an early stage of development, we foresee to have a free and publicly available dictionary for the Umbundu language available very soon.