

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

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. 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 a simplified version of the very popular note declaration syntax from the ABC Notation project[?], with some minor changes.



© 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:??

OpenAccess Series in Informatics



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

23:2 Semantics of Implicitly Timed Musical Events

45 The latter is the one we'll be discussing in greater detail.

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

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

```
53 play( 0, 100, 'A' );  
54 play( 100, 50, 'B' );  
55 play( 150, 200, 'C' );  
56
```

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

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

```
64 play( A B/2 C2 );  
65  
66
```

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

70 2 The Problem and its Requirements

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

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

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

82 Given these two requirements, we can conclude we cannot generate all music events at
83 the start and then play them in order.

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

86 Data Model

87 The basic premise is that expressions can generate a special data type: **Music**. Music is
88 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**.

Operators

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

Concatenation `Music1 Music2 ... MusicN`

Parallel `Music1 | Music2 | ... | MusicN`

Repetition `Music * Integer`

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

3 Implementation

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

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

Context State

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

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

cursor(ctx) Return the current cursor position

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

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

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

3.1 Operators

Note Events

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

23:4 Semantics of Implicitly Timed Musical Events

128 The function that emits the event receives the current context as an argument, and is
129 then responsible for creating the event, with its timestamp matching the context's cursor.
130 After the event is created, it also must move the context's cursor forward through the `seek`
131 method.

■ Listing 3 Creating a Note Event

```
132  
133 function note (ctx) {  
134     event = create_note(cursor(ctx));  
135  
136     seek(ctx, event.duration);  
137  
138     yield event;  
139 }  
140
```

141 Concatenation

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

■ Listing 4 Algorithm to concatenate musical events

```
146  
147 function concatenate (ctx, ...operands) {  
148     for (music in operands) {  
149         for (event in music(ctx)) {  
150             yield event;  
151         }  
152     }  
153 }  
154
```

155 Repetition

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

■ Listing 5 Algorithm for repetition

```
159  
160 function repeat (ctx, music, count) {  
161     for (i = 0; i < count; i++) {  
162         for (event in music(ctx)) {  
163             yield event;  
164         }  
165     }  
166 }  
167
```

168 Parallel

169 The parallel operator enables playing multiple sequences of musical events simultaneously.
170 However our events are emitted as a single sequence of ordered events, thus requiring merging
171 the multiple sequences into a single one, while maintaining the properties of laziness and
172 order. The operator assumes that each of its operands already maintains those properties
173 on their own, and so is only in charge of making sure the merged sequence does so as well.

174 With this in mind, it relies on a custom *merge sorted* algorithm for iterables (not related to
 175 the most common merge sort algorithm by John von Neumann).

■ **Listing 6** Algorithm to merge parallel musical events

```
176
177 function parallel (ctx, ...operands) {
178     for (child_ctx, event in merge_sorted(ctx, ...operands)) {
179         join(ctx, child_ctx);
180
181         yield event;
182     }
183 }
184
```

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

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

193 3.2 Integration in a Programming Environment

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

199 Variables

200 Up until now we've analyzed situations where the timing of the events is known at the
 201 moment of their creation (through inspection of the context passed onto them). However,
 202 there can be situations where the events have to be created before their time location is
 203 known.

204 We decided to take an approach to integrate this type of functionality that does not
 205 require different semantics for creating events *"live"* and for storing events in all operators.
 206 Instead, the only changes occur in variable declarations and when inserting variables in the
 207 middle of other expressions.

208 When declaring a variable, its expression is evaluated with custom context, forked from
 209 the main context, with the cursor reset back to zero. An important thing to note is that
 210 even though we are storing the music in a variable, we still take care to respect the principle
 211 of laziness present throughout our language: when the variable is declared, no actual events
 212 are created. Later, whenever the variable is used, and an event is requested of the variable,
 213 it relays that request to the expression responsible for providing the event.

214 However, one important caveat is that the variable cannot simply emit the events as they
 215 are created, because as we mentioned before, their context is a different one from the main
 216 context (with its cursor set to zero at the start of the expression).

217 Let's assume that a variable C holds the value of the `cursor(ctx)`, it `ctx` is the context
 218 where the variable is being evaluated (not declared). Then, each event would be cloned and
 219 its starting time would have the constant C added to it.

23:6 Semantics of Implicitly Timed Musical Events

220 This raises an important detail that we must not forget: many of the musical events are
221 created and then emitted directly. However, as we've seen, some can be stored in variables.
222 And variables can be used in many ways, and the same variable can be used multiple times.
223 This means that if one were to mutate an event (by change its start time or its duration,
224 for instance), we would be changing the the contents of the variable. This could be an
225 acceptable approach. But we have opted instead for a more functional approach, treating
226 those musical events as more akin to primitive values (as we would treat numbers or event
227 strings in modern languages). This means that when we change the start time of an event,
228 we are actually cloning it.

229 This works well enough because those events are very lightweight objects, and the benefits
230 of not having their values mysteriously changed outweigh the small cost of a possible
231 unnecessary allocation, in case that event was actually only used once.

232 Functions

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

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

243 In terms of managing the implicit context, functions are treated in very similar ways to
244 the regular operators. They receive the context that was active at their call site and are
245 evaluated with it. This means that any events emitted will have begin at whatever time the
246 context's cursor marks.

247 Their arguments, however, are treated the same way as variable declarations (which
248 is what they really are), meaning that each argument gets a custom fork of the call site's
249 context with its `cursor` set to zero.

250 4 Results Discussion

251 To solve our problem of keeping track of the timing implicitly for each event created, we
252 decided to pass around a context variable. There were other possible solutions, like keeping
253 this data in some sort of global variable. Our approach does give us some advantages, such
254 as being able to have multiple contexts in play at the same time. It does have drawbacks,
255 too. Every function defined in our language must receive this context to be able to create
256 events at the appropriate time. However, functions defined in Python do not expect this
257 parameter. Therefore, special conversions must be made when exchanging values between
258 both languages.

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

264 However, this early work already provides a solid foundation for a musical *DSL* that
265 while dynamic (with variables, functions and control structures) integrates very well with
266 established musical standards such as the MIDI protocol and others.