DOMAIN SPECIFIC LANGUAGE FOR COMPOSING MUSIC

Ion CERNEI*, Elena PAPUC, Dina BÎTCĂ, Cezar GUZUN

Department of Software Engineering and Authomation, Group FAF-201, Faculty of Computers, Informatics and microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Ion Cernei, ion.cernei@isa.utm.md

Abstract: *The aim of the following article is to describe the way composing music could benefit from a Domain Specific Language. It presents some music concepts and their representation using programing archetypes. Further, it explains the way the DSL works by describing it's basic commands, an example of code and parse tree.*

Key words: *music language, domain-specific language, data structures, parse tree, grammar.*

Introduction

A domain-specific language (DSL) is a computer language specialized to a particular application domain [1]. In this case the domain for the DSL is music composition, therefore most of the attention was given to computer generated sound, time management, and measurement.

Computers are general-purpose machines that can be programmed for different goals in a variety of fields, including general art and music. Computer music languages allow composers who do not have knowledge in programming, to still use computers to their advantage. Even though general-purpose programming languages can be used for composing music, experience has shown that time is a crucial aspect for this task, therefore, languages that incorporate musical time are easier to use and perform better with many musical operations. In procedural languages time can be represented with tempo, duration, abstractions of beats and schedulers [2].

The proposed language should be able to solve two problems:

1. Improving the process of learning programming by using live coding.
2. User empowerment in digital music making and the potential disruption to canonic practices of music education by the utilisation of digital technologies.

The proposed DSL is a simple and powerful way to start making electronic music and also to learn basic programming concepts.

Syntax

The term syntax, in general, refers to the notations and rules that control the structure of the programming language. The majority of computer music languages are based on text and keep the syntax similar to other programming languages. The described DSL has a simple syntax with terms as: *play, sleep, for each, repeat, function, use.*

When developind a DSL, the syntax can be text-based or graphical, but the main attention must be directed to how the music language handles timing, signals and concurrency. It can be observed that how the program behaves, it's semantics are of graeater importance than the syntax.

Semantics and language overview

The term semantics, in general, means how the programming language interprets the text. Music composition by using programming needs to include parallel processing, manually timed output, signal processing and the capability to respond to changes of the code in real-time. Therefore, innovative and interesting semantics can be found in DSLs for composing music. Programming languages for music include special data types such as signals and scores, clear specifications for timing manipulation of program behaviour and provisions for real-time interaction [2].

To create sound, the user will have the *PLAY* command, followed by a number which represents the pitch of the note. The *SLEEP* command is meant to make a timed pause between notes. After these two main commands there are those that allow the user more flexibility and less code such as *REPEAT* and *FOR EACH* commands. The *USE* command allows to choose or switch musical instruments.

To manipulate data, the user will be able to create variables and functions. There are three main reasons for using variables: communicating meaning, managing repetition and capturing the results of things.

One of the basic data structures that the DSL will have, which is very useful, is the list. A list can be declared by writing the terms within a pair of brackets, in form of a sequence where each element is separated by commas and spaces. For example [25, 30, 35].

Grammar design

For a better understanding of the grammar, special notations were used which are represented in Tab. 1.

Table 1.

Meta notations

Notations	**Meaning**
<text>	a nonterminal parameter is written between < >
text	a terminal parameter is written in bold
text*	the symbol appears zero or more times
text^{+}	the symbol appears one or more times
|	an alternative follows

There are several stages that need to be covered in order to desgn a DSL. The first stage is definition of the grammar L(G) = (V_T, V_N, S, P):

- V_T – is a finite set of characters of the Alphabet of the Grammar (terminal symbols)
- V_N – is a finite set of non-terminal symbols;
- S – is the start symbol;
- P – is a finite set of rules that combined form the production;

V_T = { **FUNCTION**, **DO**, **PLAY**, **SLEEP**, **REPEAT**, **TIMES**, **USE**, **FOR**, **EACH**, **IN**, **END**, **A**, **B**, ... **Z** , **a**, **b**, ... **z**, **0**, **1**, ... **9**, **piano**, **guitar**, **trumpet**, **drums**, **violin**, =, ., ,, **[**,**]** }

V_N = { <program>, <listOfCommands>, <basicCommands>, <initializationCommands>, <playCommand>, <sleepCommand>, <useCommand> , <repeatCommand>, <forEachCommand>, <functionCallCommand>, <naturalValue>, <floatValue>, <instrument>, <Time>, <variableName>, <listName>, <initializeFunction>, <initializeVariable>, <value>, <naturalList>, <floatList>, <functionName>, <functionBody>, <lowerCase>, <upperCase>, <digit> }

S = {<program>}

P = {

<program> →	<listOfCommands>
<listOfCommands> →	<basicCommands>$^{+}$
	| <initializationCommands>$^{+}$
	| <basicCommands> <listOfCommands>
	| <initializationCommands> <listOfCommands>
<basicCommands> →	<playCommand>
	| <sleepCommand>
	| <useCommand>
	| <repeatCommand>
	| <forEachCommand>
	| <functionCallCommand>

```
<playCommand> →           PLAY <naturalValue>
                          | PLAY <variableName>
                          | PLAY <functionName>
                          | <playCommand> <basicCommands>
<sleepCommand> →          SLEEP <time>
                          | SLEEP <variableName>
                          | <sleepCommand><basicCommands>
                          | <sleepCommand><listOfCommands>
<time> →                  <floatValue>
<floatValue> →            <naturalValue> . <naturalValue>
                          | <naturalValue>
<useCommand> →            USE <instrument>
                          | <useCommand><basicCommands>
<repeatCommand> →         REPEAT <naturalValue> TIMES <basicCommands> END
                          | <repeatCommand><basicCommands>
<forEachCommand> →
          FOR EACH <variableName> IN <listName> DO <basicCommands> END
          | <forEachCommand><basicCommands>
<functionCallCommand> →         <functionName>
                                | <functionCallCommand> <basicCommands>
<initializationCommands> →      <initializeFunction>
                                | <initializeVariable>
<initializeVariable> → <variableName> = <value>
<value> →            <naturalValue>
                     | <floatValue>
                     | [<naturalList>]
                     | [<floatList>]
<naturalList> →      <naturalValue> , <naturalList>
                     | <naturalValue>
<floatList> →        <floatValue> , <floatList>
                     | <floatValue>
<initializeFunction> →     FUNCTION <functionName> DO <functionBody> END
<functionBody> →          <basicCommands>
< naturalValue > →        <digit>+
<variableName> →          <lowerCase>+ | <upperCase>+ | _+ | <digit>+
<functionName> →          <lowerCase>+ | <upperCase>+ | _+ | <digit>+
<listName> →              <lowerCase>+ | <upperCase>+ | _+ | <digit>+

<lowerCase> →        a | ... | z
<upperCase> →        A | ... | Z
<digit> →            0 | ... | 9
<instrument> →       piano | guitar| violin | drums
}
```

Code example

The following code shows an example of function and variable declaration, also play, sleep, for each, and repeat commands.

```
function simpleNote do:
  play 50
  sleep 0.5
end
```

```
notes = [50, 60, 80]
use piano
for each note in notes do
   play note
   sleep 0.5
end
use violin
repeat 3 times:
   simpleNote
end
```

Parse Tree

In Fig. 1 it is represented the Parsing Tree of the code from above. The first branch shows the function initialization command and the second branch shows other basic commands.

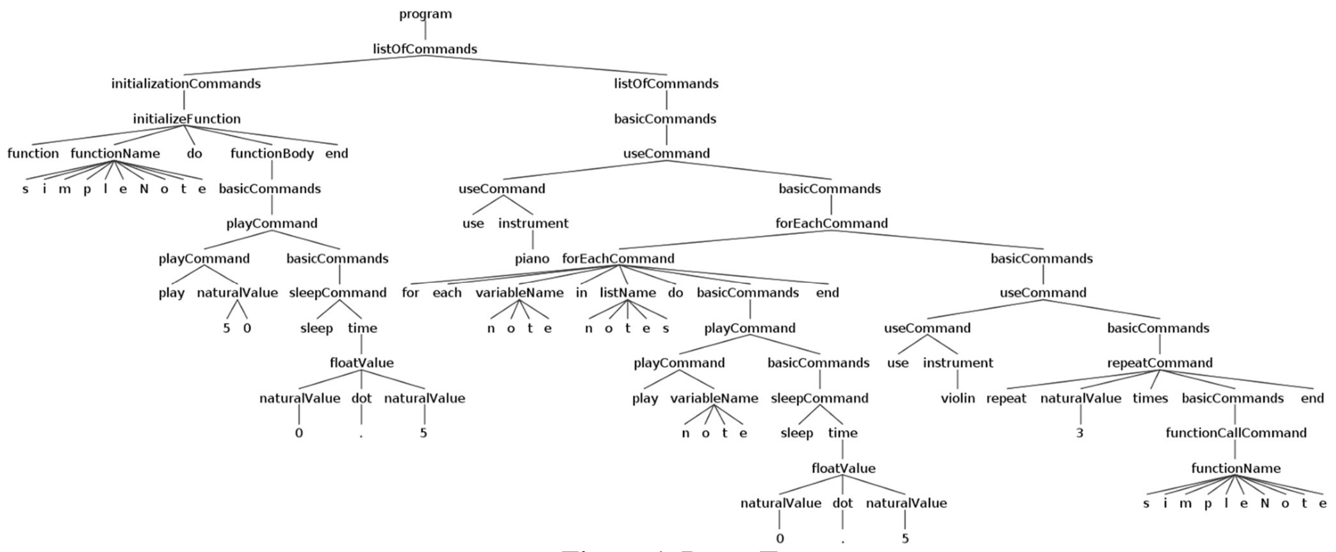

Figure 1. Parse Tree

Conclusions

Domain-specific languages for music are formed by techniques and ideas that can recreate the original prcess of music composition and make it simpler. The difference between Music DSLs and other languages is that they must deal with the concept of time, simultaneous processes, and audio signals. These concepts are naturally understood by humans when it comes to composing music traditionally, but they can be hard to elaborate in conventional programming languages. Since composing music is a process more bound to creativity than to engineering rules, it is important for languages to provide a way for quick experimentation, therefore the syntax and semantics should be adapted for this as well.

References

1. PARSONS, REBECCA. *Domain Specific Languages,* In Google Scholar, 2019, pp. 89-94.
2. DANNENBERG, RB. *Languages for Computer Music,* In Google Scholar, 2018, pp. 2-11.
3. JOSEPH A. GOGUEN. *Semantics of computation*, Lecture Notes in Computer Science, Vol. 25, Springer, 1975, pp. 151–163