DOMAIN SPECIFIC LANGUAGE FOR SOUND PROCESSING

Mihail ECHIM[1*], Anastasia CUNEV[1],
Iulian BERCU[1], Stefan NISTOR[1]

[1]Department of Software Engineering and Automation, FAF-211,Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Moldova

*Corresponding author: Mihail Echim, mihail.echim@isa.utm.md

Scientific coordinator: Gabriel ZAHARIA, University Assistant, Technical University of Moldova

Abstract: *This article relays an analysis of the implementation of a Domain-Specific Language for sound processing. In this article, both the technical and the non-technical aspects of the Domain-Specific Language implementation were analyzed and it was explained in detail the process and the steps that were taken in order to develop the Domain-Specific Language for Sound Processing, as well as its rules and specifics were conveyed.*

Key words*: DSL, language, grammar, semantics, syntax, sound processing*

Introduction

Programming presents a wide variety of problems that programmers solve using various existing tools. However, sometimes a new tool is developed to make certain tasks easier such as a Domain-Specific Language (DSL). A DSL presents tools for certain specific tasks that prove more useful and easy to use than General-Purpose Languages (GPL), because DSLs are created with a specific focus in mind to accomplish certain tasks as frictionlessly as possible [1].

When creating DSLs, programmers have the opportunity to shape the tool to their needs as closely as possible which can greatly speed up development processes on bigger projects.

Additionally, DSLs can provide a higher level of abstraction than GPLs, which can make them more approachable to non-programmers or those with less programming experience. This can be especially beneficial in fields where technical knowledge is not the primary focus, such as finance, biology, or law [2]. A DSL is a user empowerment tool for increasing software system development productivity. By creating a DSL that speaks the same language as the experts in those fields, programmers can empower those experts to more easily translate their knowledge into code and automate repetitive tasks. Ultimately, DSLs provide a powerful tool for solving specific problems in a more intuitive and efficient way, and their development can greatly benefit both programmers and non-programmers alike [3].

Domain analysis

The problem that has to be solved is creating a DSL that provides a more intuitive and concise way of expressing algorithms for sound processing, making it easier for practitioners and researchers in the sound processing field to express their ideas and techniques.

The problem solution also involves the creation of a special syntax and constructs for improved readability, especially for those familiar with the sound processing domain. And in general, it will be able to do all the general sound processing techniques such as filtering, spectral analysis, and compression.

A DSL is one of the best solutions for these problems because DSLs provide a more intuitive and concise way of expressing sound processing algorithms, as compared to general-purpose programming languages. This leads to increased productivity and improved readability of code. In addition, DSLs can also provide domain-specific optimizations, leading to improved performance and scalability.

For the project, the research was conducted in the field of sound processing to explore and analyze the existing DSLs. Some popular DSLs for sound processing include Max/MSP, Pure Data, and SuperCollider [4]. The plan of the project is to look at all the advantages that the languages provide and one up them to create a truly intuitive and convenient sound processing DSL.

The project is specifically meant for audio engineers who often work with complex sound processing algorithms, and a DSL can provide a more intuitive and efficient way of expressing these algorithms. Music producers who use it for sound processing to quickly create and manipulate sounds, without having to write complex code. Researchers in the fields of audio signal processing and music technology who can use a DSL to prototype and evaluate new sound processing algorithms and techniques. And finally, people interested in music production who want to learn code in a fun and interactive way.

Grammar

In Table 1 are presented special notations, which are useful for better understanding of the grammar.

Table 1

Meta Notations

Notations	Meaning
<x>	x is a nonterminal symbol
x	x is a terminal symbol (text in **bold** font)
[x]	means zero or one occurrence of x
x*	means zero or more occurrence of x
x^+	means one or more occurrence of x
|	separates alternatives

To design a grammar, there are several tuples that need to be implemented. Every Grammar consists of 4 tuples as follows G = (V_T, V_N, P, S), where:

1. V_T - is a finite set of terminal symbols.
2. V_N - is a finite set of non-terminal symbols.
3. P - is a finite set of productions of rules.
4. S - is a start symbol [5].

S = {<program>}

V_T = {**0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, …, x, y, z, A, B, C, …, X, Y, Z, true, false, +, -, /, *, Sine, Sawtooth, Triangle, Square, lowpass, highpass, distortion, phase, time, envelope, play, record, volume, pan, reverb, gain, freq, :, ;, (,), >, <, }, {, “, func process(), func compose(), if, else, for, return, break, continue, >, >=, ==, !=, <, <=, //**}

Vn = {<program>, <statement>, <query_invocation>, <comments>, <variable_declaration>, <method_invocation>, <assignment_statement>, <identifier>, <value>, <values> <characters>, <character>, <string>, <digits>, <digit>, <non_zero_digit>, <method_name>, <comment>, <method_declaration>, <expression>, <term>, <operation>, <arithmetic_operation>, <comparative_operation>, <variable>, <constant>, }

P = {

<program> -> **func process**() {<statement>$^+$}* **func compose**() {<statement>$^+$}

<statement> -> <query_invocation>
| <comments>
| **if** (<expression>) <statement>**else** <statement>
| **for** <id> **=** <expression> **,** <expression> <statement>
| **return** <expression>

| **break**
| **continue**
| <statement>
<query_ivocation> -> <variable_declaration>
| <method_invocation>
| <method_declaration>
| <assignment_statement>
<expression> -> <term>| <expression> <operation> <expression> | (<expression>)
<term> -> <variable> | <constant>
<variable> -> <identifier>
<constant> -> <digits>
<operation> -> <arithmetic_operation>| <comparative_operation>
<arithmetic_operation> -> **+**| **-**| *****| **/**
<comparative_operation> -> **>**| **>=**| **==**| **<**| **<=**| **!=**
<variable_declaration> -> <identifier>=<value> | <identifier> = <method_invocation>
<identifier> -> <characters>| <characters><string>
<values> -> <value> | <value>,<values>
<value> -> <digits> | <characters>
<digits> → <digit> |<digit><digits>
<digit> → **0**| <non_zero_digit>
<non_zero_digit> → **1**| **2**| **3**| **4**| **5**| **6**| **7**| **8**| **9**
<characters> -> <character> |<character><characters>
<string> -> **'**<characters>**'** | **'**<digits>**'**
<character> → **a**| **b**| **c** … **x**| **y**| **z** … **A**| **B**| **C**| … **X**| **Y**| **Z**
<method_invocation> -> <method_name>([<values>$^{+}$])
<method_name> -> load| delay| set_gain| set_reverb|...
<assigment_statement> -> <identifier> = <value>
| <identifier> = <identifier>
| <identifier> = <string>
<comments> → <comment>| <comments><comment>
<comment> → // <string> }

Code Example:

```
func process(){
        mysound = load(sound.mp3)
        setreverb(15 mysound)
        delay(mysound)
}
func compose(){
        play(mysound)
}
```

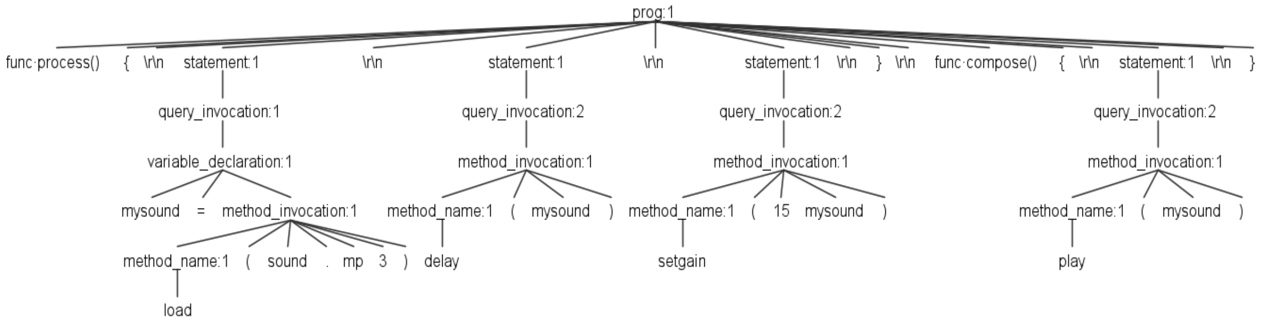

Figure 1. Parse Tree

Semantics and lexicon

In the first block of the program, the user will be able to manipulate and modify sounds. This can include changing the pitch, adding effects such as reverb or distortion, or chopping up the sound into smaller samples. The DSL will provide a variety of tools and options for the user to experiment with, allowing them to create unique and interesting sounds. Additionally, the program may allow the user to import their own audio files to use in their compositions, giving them even more flexibility and creative control.

Once the user has created and modified their sounds in the first block of the program, they will move on to the second block to compose their track. In this block, the user will be able to place their sounds on a timeline, specifying the exact moment when each sound should be played. They may also be able to adjust the volume and other parameters of each sound within the track, and add additional effects or transitions between sounds. As the user works on their composition, they may be able to listen to the track as it is being built, allowing them to make adjustments and refine the sound until it is exactly what they are looking for. Ultimately, the DSL will allow the user to express their creativity and musical ideas in a tangible way, creating a unique and personalized composition that is entirely their own.

Conclusions

In conclusion, this project aims to create a DSL for sound processing that provides a more intuitive way to express sound processing algorithms. DSLs are designed with a specific focus in mind and can provide easy-to-use tools. By creating a DSL, programmers have the opportunity to customize the tool to their specific needs, which can significantly speed up the development process for larger projects. Compared to General Purpose Language, a Domain Specific Language can provide a higher level of abstraction, which can make them more approachable to non-programmers or those with less programming experience.

References

1. FOWLER, M., PARSONS, R., *Domain Specific Languages* [online][accessed 03.03.2023]. Available: *https://martinfowler.com/books/dsl.html*
2. VOELTER, M., *DSL Engineering Designing, Implementing and Using Domain-Specific Languages* [online]. [accessed 09.03.2023]. Available: http://voelter.de/data/books/markusvoelter-dslengineering-1.0.pdf
3. BARKATI, K., JOUVELOT, P., *Synchronous programming in audio processing* [online]. [accessed 07.03.2023]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-01540047/document
4. BARISIC, A., GOULÃO, M., AMARAL, V., *Usability Driven DSL development with USE-ME* [online]. [accessed 09.03.2023]. Available: https://hal.science/hal-03159941/file/Bari%C5%A1i%C4%87,%20Amaral,%20Goul%C3%A3o%20-%202018%20-%20Usability%20Driven%20DSL%20development%20with%20USE-ME-annotated.pdf
5. COJUHARI, I., DUCA, L., FIODOROV, I., *Formal Languages and Finite Automata* [online]. [accessed 03.03.2023]. Available: https://else.fcim.utm.md/pluginfile.php/110458/mod_resource/content/0/LFPC_Guide.pdf