DrumLace – A Domain Specific Language (DSL) for Drum Programming

André Semanas de Oliveira Araújo
ALGORITMI Research Centre/LASI – DI, University of Minho, Braga, Portugal

José João Dias de Almeida
ALGORITMI Research Centre/LASI – DI, University of Minho, Braga, Portugal

Pedro Rangel Henriques
ALGORITMI Research Centre/LASI – DI, University of Minho, Braga, Portugal

Abstract

In an ever expanding world of musical technology, and due to the evolution of computational power in the last decades, a subdomain emerged where software represents a tool for musicians. Within that subdomain a large variety of approaches exist that allow users to create, define, improve and analyze music. One of these approaches involves the development of text-based languages, that allows the user to define music only through text in a way that enables the development of tools which make possible the printing of the musical sheet describing the score, as well as the generation of different audio files – in WAV, MP3, or other formats – that other music software, such as software synthesizers, can use to play the described score.

This project aims to develop a new language, focused on percussion instruments, that allows the description of rhythms, via text or via a visual language – made available trough a Graphic User Interface (GUI). Moreover, that new music DSL supports the use of functions applied to the rhythms and allows the generation of various outputs. This narrow focus on drum programming, as opposed to music as whole, aims at providing an easier to learn syntax, a simple to use tool and an environment open to the integration of more complex concepts presented in the world of percussion.

2012 ACM Subject Classification Applied computing → Sound and music computing; Software and its engineering → Specialized application languages; Software and its engineering → Compilers

Keywords and phrases Domain Specific Languages (DSL), Visual Musical Languages, Drum Programming, Compilation

Digital Object Identifier 10.4230/OASIcs.SLATE.2024.8

Category Short Paper

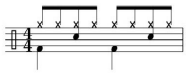
Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

1 Introduction

As long as there has been music, there have also been tools to develop and perform it, and as technology evolves so do those tools. In the 19th century, while discussing the possibilities of the analytical engine, a predecessor of the modern computer, a mathematician named Ada Lovelace proposed that such a machine could be used to program music [12]. Max Mathews made that vision a reality in the 1950's with the creation of the first programming language MUSIC I [4]. Now, thanks to their work, the world of music-related software, including but not limited to programming languages, is a fascinating, complex and rich environment in which new ideas blossom every day.

Although the creation of domain specific languages applied to music is nothing new (as will be detailed in Section 2.1), the ones already existing seem insufficient. They lack a reasonable accessibility for most users. This is due to the fact that most languages usually encompass all

```
%%beginsvg
<defs>
<path id="x_head"
d="m-3 -3l6 6m0 -6l-6 6"
class="stroke" style="stroke-width:1.2"/>
</defs>
%%endsvg
X:1
M:4/4
L:1/4
%%score (1 2)
K:C clef=perc
%%map drum ^A, print=g heads=x_head
%%map drum D, print=c
%%map drum C, print=F
[V:1] [I:voicemap drum][I:MIDI channel 10]
^A,/^A,/[^A,D,]/^A,/ ^A,/^A,/[^A,D,]/^A,/ |
[V:2] [I:voicemap drum][I:MIDI channel 10]
C,xC,x |
```



```
\version "2.24.2"
plates =
\drummode {
hh8 hh hh hh hh
hh hh hh
}
bdns =
\drummode {
bd4 sn bd sn
}
\score{
\new DrumStaff
<<
\new DrumVoice
{\voiceOne
\plates}
\new DrumVoice
{\voiceTwo \bdns}
>>
\layout{}
}
```

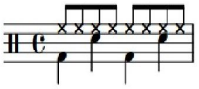


Figure 1 Example of the same rhythm defined in ABC and LilyPond notation.

areas of music what requires that those languages have a high degree of expressiveness, which usually comes at a cost of simplicity and easiness of use. The lack of a carefully designed GUI (graphical user interface), that helps the edition of such musical sentences, also makes them less appealing to users without a programming background. These evidences motivated us to carry on the development of DrumLace system here discussed.

The main goal of this project is to create a more specific language than the existing ones, dedicated only to rhythm programming of percussion instruments, also known as drum programming. By targeting a more focused subdomain of music, and devising strategies that enhance the user accessibility, it should be possible to make a more comprehensible and easy to use language, thus providing not only a platform for the simple descriptions of drum patterns but a framework for the creation of complex rhythms that would otherwise be difficult to describe.

To introduce DrumLace programming language and its processor, that allows musicians to write and play drum scores, the paper is organized in five sections. Section 2 presents the state-of-the-art concerning programming languages and software tools for music. Section 3 is the core of the paper, as it introduces DrumLace project, proposing the system architecture, defining formally the DSL designed and sketching the planned GUI. Section 4 illustrates the proposal with concrete examples showing outputs actually produced by DrumLace compiler. Section 5 closes the paper with some conclusions sorted out from the development of DrumLace project, and proposing directions for future work.

2 Music Programming Languages and Music related Software

The description of the state-of-the-art concerning the area of software to support musicians in their creative or training activities, will be divided into two parts. The first one focuses on text-based languages, while the second introduces and discusses some features of specific music software that are relevant for the development of DrumLace language.

2.1 Programming languages

A music programming language is a domain specific language specially tailored for the musical domain, allowing for describing a score. Examples of well known music programming languages are LilyPond[9] and ABC[2].

Both ABC and LilyPond describe a music score in a way that an interpreter is able to generate sheets, MIDI files and sound files. Some key differences are the existence of a drum mode in LilyPond, that handles the MIDI mapping for drum programming, and, in contrast with ABC notation, LilyPond contains the instructions for interpretation - the format of the output - within the text file itself. ABC relies on external programs such as abcm2ps or abcmidi.

The formal definition of such languages enables the design and implementation of tools able to parse and understand the scores written in those languages in order to produce different outputs, such as the printed sheet of the score written to a PDF or SVG file, or the corresponding sound written to different file formats compatible with electronic devices capable of playing the music.

2.2 Relevant Music Software

There are many approaches to music software[7]. This section will focus on certain features present in different paradigms .

Firstly, the Digital Audio Workstation (DAW). A DAW works as a virtual studio[6]. A prominent feature among most if not all DAWs is the use of a piano roll[10], which is a visual component that allows for the sequencing of different notes, or in the case of drum programming, different instruments on a Matrix that will then be interpreted in terms of time.

Another software approach worth mentioning is the patching paradigm in which different processing blocks are connected to each other allowing for a pipeline of events. In the back end of the GUI is a programming language that is accessible to the user which can read and alter the code. This approach allows for the ease of use of a GUI and the expressiveness of a text based programming language and is present in software such as MaxMsp[5].

Lastly, it is important to mention how sound is actually made. This is done trough the the MIDI format[3], which consists on information regarding musical events such as notes, their pitch or volume, or any other relevant parameter. MIDI events are then interpreted by synthesizers which produce sound in many forms[11]. It is important for this work to highlight sample-based synthesis[1] in which MIDI events trigger certain pre-recorded sounds. This is the case for fluidsynth[8].

3 DrumLace

In order to achieve the goals mentioned in Section 1 it is clear that not only a text language is needed, but also a system that generates and processes text in a specific manner.

The DrumLace language, named after Ada Lovelace, is a system developed with the goal of creating a framework for drum programming, that achieves both the accessibility of a graphic interface as the expressiveness of a programming language. To reach that balance, it is only necessary that a GUI is capable of “writing” text in the DrumLace text language.

3.1 System Architecture

The chosen implementation was to allow a graphic description of rhythms that would then be converted into a text file written according to the DrumLace grammar, present in section 3.3. After the text file is produced, it can be edited by the user to either change what as been described or add additional functions, descriptions or exporting options. Finally, the text file is processed as explained in Section 4, thus producing the desired output, according to the export options. Figure 2 shows the architecture of DrumLace system.

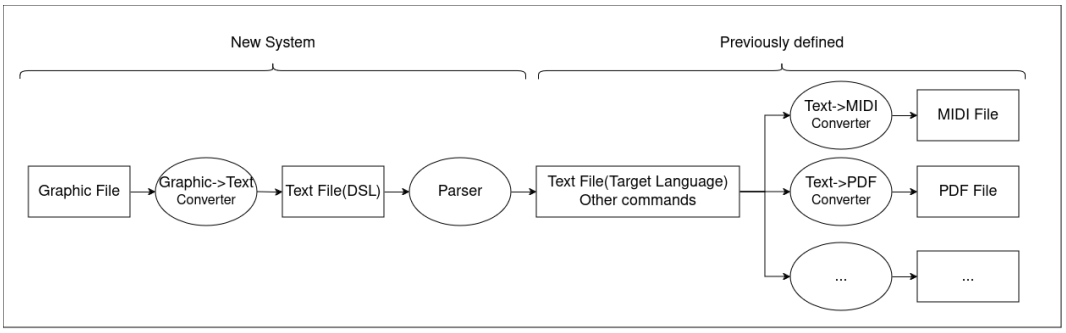


Figure 2 DrumLace processing flowchart.

3.2 Rhythm Algebra and Functions

A worthwhile feature of our language is the ability to apply a certain algebra of rhythms that gives the user the option of not only create patterns but combine them or alter them in different ways.

That goal is achieved by employing an algebra of rhythms, which can be seen in Figure 3, while also having the option of applying functions. For example,

```
DP a = #drum pattern description
DP b = rev(a)
```

is a fragment of a DrumLace program that declares two drum patterns (DP), `a` and `b`, and uses a reverse function (`rev`) which is applied to the drum pattern `a`. that can have different effects, such as an horizontal inversion of a pattern.

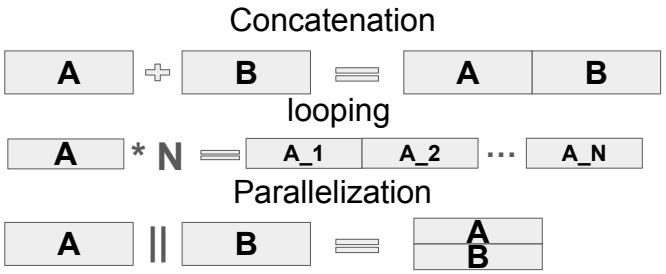


Figure 3 Drum Pattern algebra.

Figure 3 depicts the three operators of the algebra that we are considering in our Drum Pattern language. The first one represents the *concatenation* of two drum patterns in which pattern A will be followed by pattern B. The second one, the *looping* operator, represents the repetition of a pattern A n times. The third one represents the *parallelization* of two patterns; it allows for the creation of a new pattern that consists on both A and B at the same time.

3.3 Grammar

A crucial part of DrumLace system, is the grammar which dictates the rules for writing a drum score (a musical program). This context free grammar (CFG), as showcased in the listing below, states that a program (denoted by the non-terminal symbol `start`) is made up

from two distinct parts: assignments (`asgs`), and transformations (`exports`). An *assignment* is a statement (a language construction) by which a drum pattern, identified by a unique name (an `ID`), is given a *value* that is drum pattern *expression* built according to the algebra of Figure 3. A *transformation* (or an `export`) is a command, another program statement, that instructs the system what to produce from those drum patterns previously declared in the assignment part.

```
start : asg* export*
export : ("export"i "(" ID ex_args")")          -> export
        |("Play"i "(" ID ")")                   -> play
ex_args: EX_ARG("," EX_ARG)*                    -> export arguments
asgs : ID "=" cexp                              -> drum pattern assignment
cexp:pexp
    |cexp "+" pexp                              -> concatenation
pexp:alg_exp
    |pexp "||" alg_exp                          -> parallelization
alg_exp:ID                                      -> identity
        |drum_pattern                           -> pattern_asg
        |ID "(" func_arg ("," func_arg)*")"     -> function
        | alg_exp "*" NUM                       -> looping
        | "(" alg_exp ")"                       -> grouping exp
func_arg:alg_exp                                -> functions argument
        |NUM
drum_pattern : "{" TIME_SIG ";"
                 "Tempo" "=" TEMPO ";"
                 instrument_lines "}"            > drum_description
instrument_lines : (INST notes* ";")+           -> seq of instruments id
notes : ("1" "/" NOTE_LENGTH "|" note_seq "|")* -> sequence of notes
note_seq : (HIT|PAUSE)*                         -> note value
```

The CFG defined in the listing above is written in Lark metalanguage, using extended-BNF notation[1]. Non-terminal symbols are denoted by identifiers in lower case, while terminals are written in upper case. Keywords and one-character signs are written inside quotes. Notice the small "`i`" immediately after a keyword means that the language is case-insensitive. Each production is associated with a multi-word identifier using on the right side the operator "`->`".

3.4 DrumLace Interface

Finally, Figure 4 shows a mock-up of a pattern being described in a graphic interface, which then generates a text file with the description of the pattern. Then the description is followed by some functions and algebraic expressions, and two export instructions. Resulting in the described rhythm being played and the creation of a PDF file with the sheet allowing other musicians to read it.

Note that a pattern assignment in the actual language does not need to be preceded by a "DP". In this paper, until now, this was only used to clarify a drum pattern (DP) assignment.

[1] Notice that the iterative operator start, "`*`", is used in the RHS of productions to represent the repetition of a sequence of symbols zero or more times, instead of recursion.

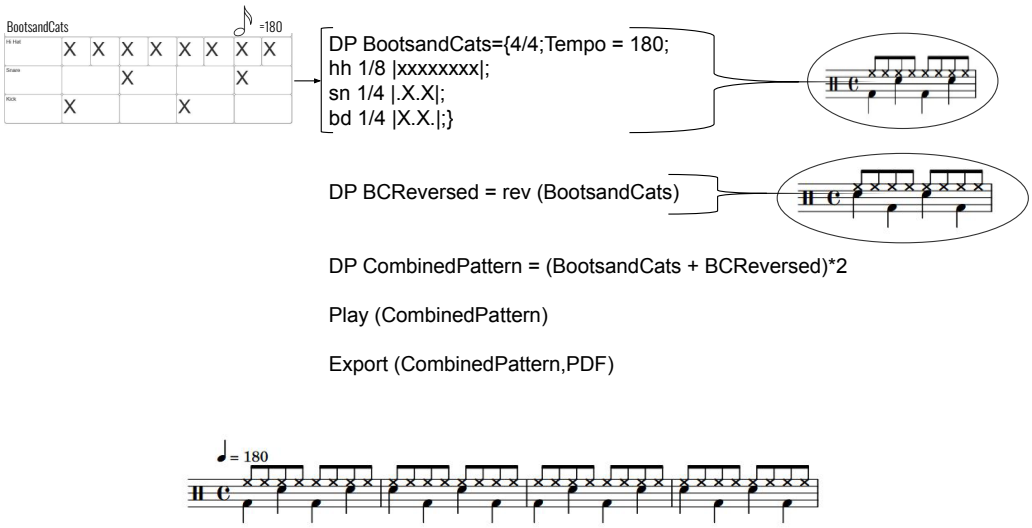


Figure 4 DrumLace mock-up.

4 Using DrumLace, an example

To better understand DrumLace programming language and the functioning of its compiler, it is important to analyze and interpret the result of processing of a DrumLace program.

Firstly, the abstract syntax tree (AST) is generated by a parser developed in Lark that follows the grammar presented in Section 3.3. This AST is the internal representation of the input music program and it describes the structure of the program and contains all of the relevant information, ignoring any comments or blank space that may be left by the user.

To illustrate this process, consider the flowing DrumLace input program:

```
pattern={4/4;Tempo=180;
          bd 1/4 |X.X.|;
          sn 1/4 |.X.X|;}
export(pattern,'./out',PDF)
```

The AST built by our parser, after analysing the input, is shown in Figure 5.

After the AST is created, the new drum patterns are calculated and the program is then divided into two lists, one containing the information for each drum pattern, the other one containing the export stack, which describes what to do with them (see Figure 6).

Notice that the Play instruction is a special case of an export in which the output target and the file format is implied. Additional processing steps will be required to actually play the pattern. Because a single LilyPond file can't create various patterns and then generate different files from them, each export creates a target LilyPond file and a Shell script that, according to the export in question, compiles said target file and, if needed, further processes the result of the first operation. Thus generating, from the AST, the desired outputs. This internal organization is illustrated in Figure 6.

To illustrate the use of DrumLace musical language and compiler in its present stage, we present below two complete examples.

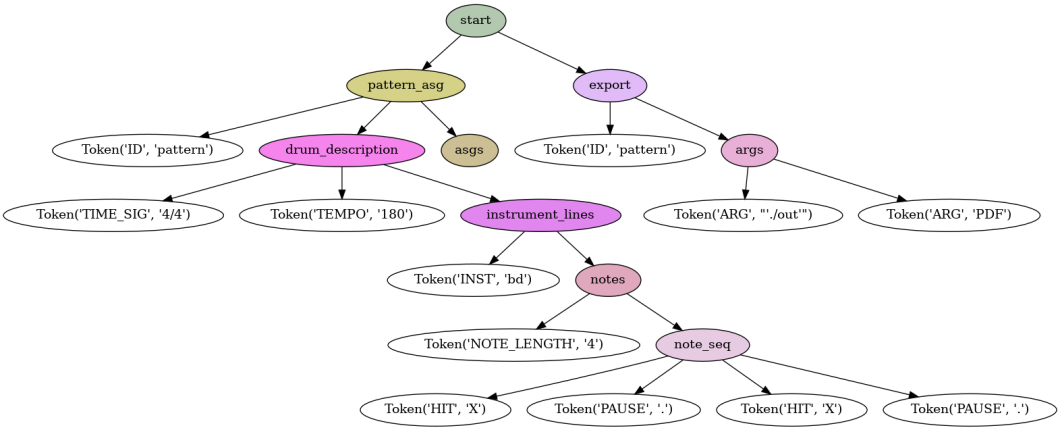


Figure 5 The AST built after parsing a DrumLace program.

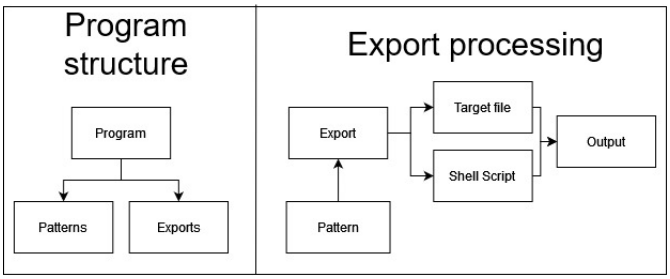


Figure 6 Program structure and Export Processing.

The first one, shown in Figure 7, starts with two drum descriptions, assigning them to the DP names `pattern` and `patterntwo`, and exports both to a PDF file to create the respective sheets (displayed on the right side of each description). Moreover, the second drum pattern is also export to a WAV file to play the score.

The second example, is the program bellow:

```
a={4/4;Tempo=180;
      bd 1/4 |X.X.|;sn 1/4 |.X.X|;
      hh 1/16 |X.X.XXX.X.XXX.X|;}
b = rev(a)  #reverse function
c={4/4;Tempo=180;
      tomfl 1/8 |XX.X..X.|;hh 1/8 |XXXXXXXXX|;
      bd 1/8 |.X.X.X.X.|;}
result = (a+b)*2 + c
  export(result,'./out',PDF)
  export(result,'./out',WAV)
```

This DrumLace program also declares two drum patterns, now named `a` and `c`, and uses the algebraic operators and functions to build expressions to construct new patterns, now named `b` and `result`. After creating the three DP, the pattern `result` is exported twice. First to a PDF file to produce its sheet (shown in Figure 8); the second to a WAV file to play the programmed rhythm.

As it is not possible to incorporate sound files in this document, the resulting WAV files – named `patterntwoWAV.wav` for the first example, and `resultWAV.wav` for the second example – are published and accessible in the anonymous github repository : `https://github.com/DLFilesShare/DrumLace-slate-24`.

```
pattern={4/4;Tempo=120;
bd 1/4 |X.X.|;
sn 1/4 |.X.X|;
}

patterntwo={4/4;Tempo=180;
hh 1/8 |X.X.X.X.|;
sn 1/4 |.X.XXX|;
tomfl 1/8 |X..XX..X|;
cymc 1/2 |...|1/1 |X|;
}

export(patterntwo,'./out',WAV)
export(patterntwo,'./out',PDF)
export(pattern,'./out',PDF)
```

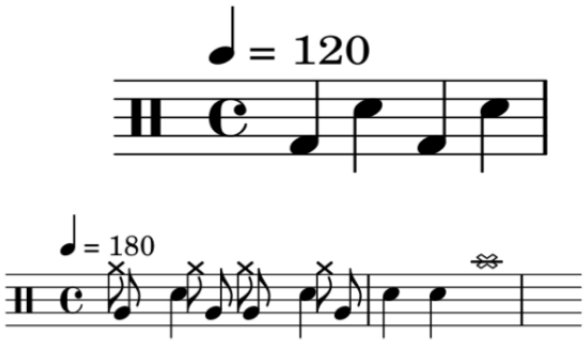


Figure 7 Example of two drum descriptions in DrumLace and their respective sheets printed in PDF.

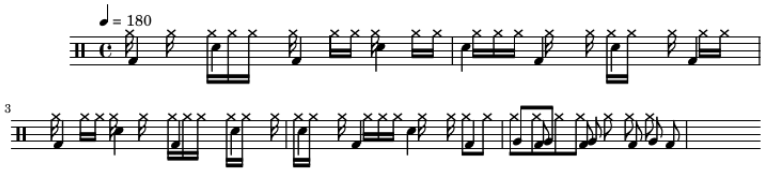


Figure 8 Example of a drum rhythm defined through functions and algebraic expressions applied to drum patterns.

5 Conclusion

Along the paper we proposed and discussed the design and development of a simple DSL for drum rhythms programming.It was also presented a compiler that produces, from the input, both a musical sheet describing the score and a LilyPond file to play it.

Sections 3 and 4 provided the details about the grammar that defines the referred DSL, and about the processor developed to recognize the rhythm sentences and to generate the desired outputs. The design of DrumLace language took into considerations the existing music languages and related tools (summarized in Section 2) to maximize the user experience.

By narrowing the focus to drum programming, the mappings and other obstacles of other music programming languages were overcame, and by allowing the use of algebraic expressions and functions the expressiveness of the DrumLace language allows for a myriad of new possibilities.

Even though much is done, there are still many possible features that can be added, with the most prominent one being the GUI that is still in a very early stage of development.

Other possible features include the programming of live playable samplers that allow for the triggering - trough a keyboard - of different rhythms, the inclusion of control patterns that apply functions or alterations to patterns in a cyclical fashion, irrational rhythms and new functions as well as the ability to define new ones. Another task of uttermost relevance is the design and conduction of one or more experiments with real users to assess the overall performance of the system, its usability, and the user satisfaction.

References

1 Sample-based synthesis. https://support.apple.com/pt-pt/guide/logicpro/lgsife418f0c/mac.

2 Guido Gonzato. *Making Music with Abc 2 A practical guide to the Abc notation*. sourceforge, 2022.

3 Jim Heckroth. A tutorial on midi and wavetable music synthesis. *Application Note, CRYSTAL a division of CIRRUS LOGIC*, 1998.

4 Max Mathews/Bell Labs. Music, 1957.

5 Vincent J Manzo. *Max/MSP/Jitter for music: A practical guide to developing interactive music systems for education and more*. Oxford University Press, 2016.

6 Mark Marrington et al. Composing with the digital audio workstation. *The singer-songwriter handbook*, pages 77–89, 2017.

7 Andreas Möllenkamp. Paradigms of music software development. In *Proceedings of the 9th Conference on Interdisciplinary Musicology–CIM14*, pages 61–63, 2014.

8 Jan Newmarch and Jan Newmarch. Fluidsynth. *Linux Sound Programming*, pages 351–353, 2017.

9 Han-Wen Nienhuys-Jan Nieuwenhuizen and H Nienhuys. Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (konferenciaanyag)*, 2003.

10 Anders Reuter. Who let the daws out? the digital in a new generation of the digital audio workstation. *Popular Music and Society*, 45(2):113–128, 2022.

11 Martin Russ. *Sound synthesis and sampling*. Routledge, 2012.

12 Ge Wang. A history of programming and music, 2012.