1 e + a: Language Specification

Valeria Starkova

Spring 2022

1 Introduction

This language allows drummers to notate drums efficiently, without using inconvenient intermediary software. As it can be tedious to write drum tabs in UIs that are not designed for this purpose, this language lets users write drum tabs in a simple and intuitive way: by utilizing the good old "1 e and a" counting system.

A drum part's structure is often constant throughout a song, with only slight variations across its parts (intro, verse, pre-chorus, chorus, bridge, etc.). This language also helps to avoid repetition by allowing the user to write one structure only once, then reuse it in other parts with slight modifications if necessary.

2 Design Principles

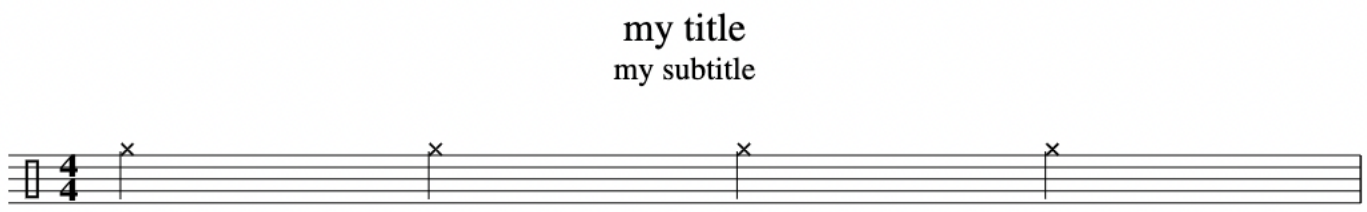
The most important design principle in this language is the fact that users can create and re-use patterns, bars, and beats that can be easily modified when writing the tabs. Perhaps in the future, users will be able to share the source code for each song written in this programming language on a platform similar to GitHub, where they can keep track of their commits, collaborators, etc. Further, the platform will contain a built-in editor and a visual display of the tabs that can be directly played.

3 Examples

3.1 Example 1

It is very easy to create a pattern, just by using the "1 e and a" counting system. We put separators `|` to disambiguate the division of the beats.

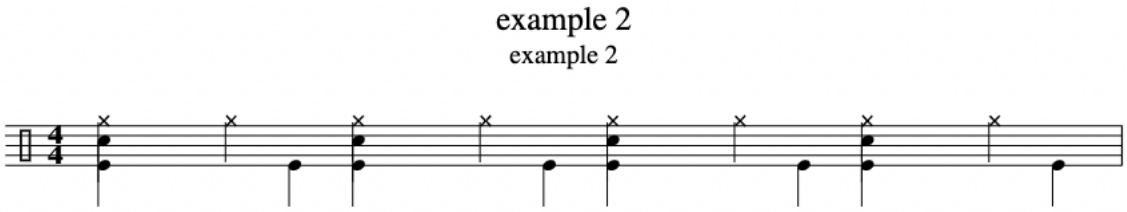
```
time: 4/4
division: 1/16

title: my title
subtitle: my subtitle

pattern mypattern : 1 | 2 | 3 | 4 |

render: mypattern
```



3.2 Example 2

Simple example of creating a one measure beat.

```
time :  4/4
division: 1/16

title: example 2
subtitle: example 2

pattern mypattern : 1 | 2 | 3 | 4 |


bar mybar1:
    hh: [ 1 + | 2  + | 3 + | 4  + | ]
    sn: [ mypattern ]
    bd: [ 1 a | 2 a | 3 a | 4 a |]


render: mybar1
```

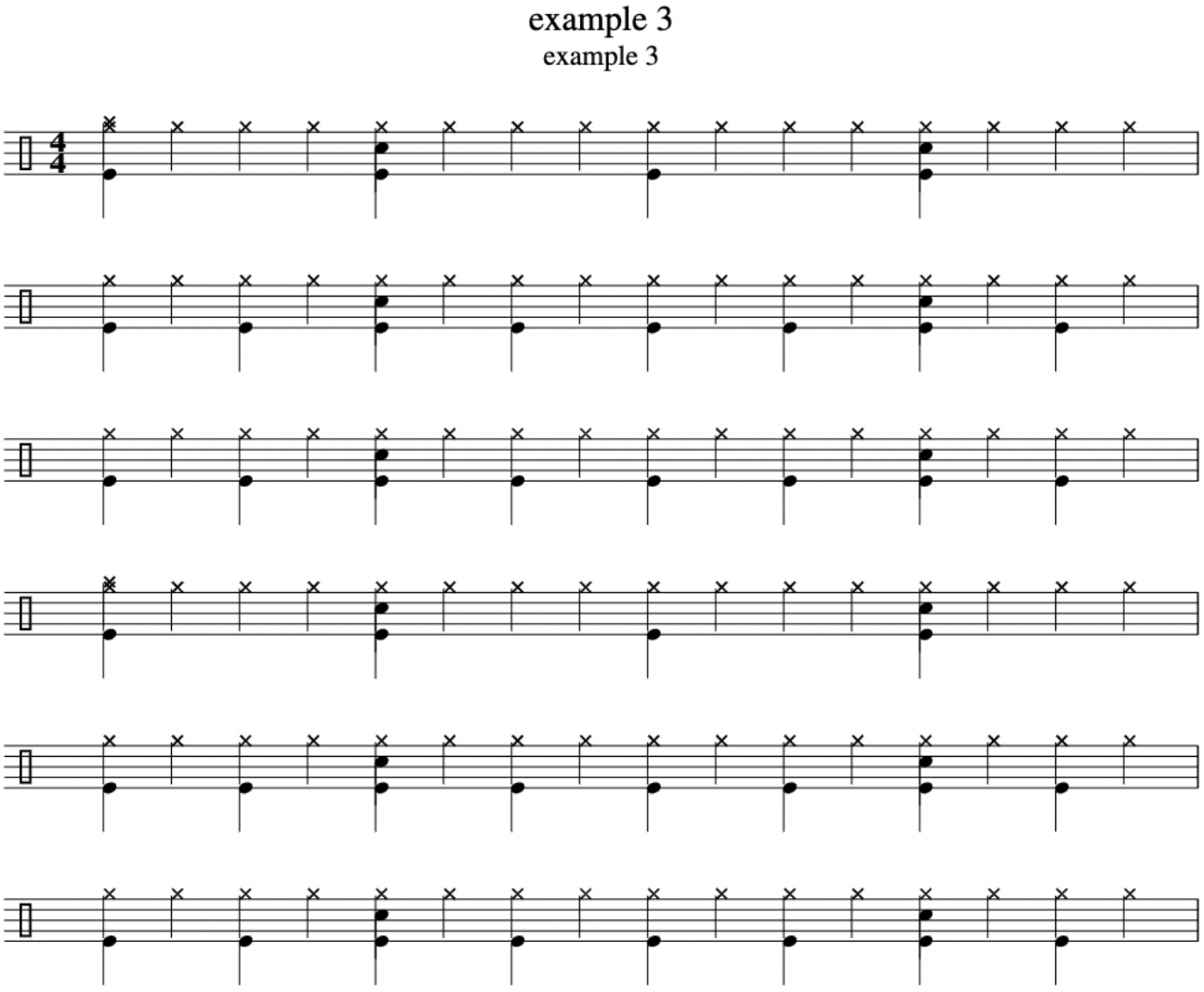


3.3 Example 3

Create a beat by repeating the pre-defined measure 4 times and adding a crash cymbal on the first beat and changing the pattern of the bass drum. Also repeat the bar 2 times without any change instructions:

```
time :  4/4
division: 1/16

title: example 3
subtitle: example 3

pattern mypattern : 1 e + a | 2  e + a  | 3 e + a  | 4  e + a  |

bar mybar1:
    hh: [ mypattern ]
    sn: [ | 2 |  | 4 |]
    bd: [ 1 + | 2  + | 3 + | 4 + |]
```

```
Snippet mysnippet:

    @change 4:
        mybar1 (1,4) {
            cc: [1 |]
            bd: [ 1 | 2 | 3 | 4 |]
        };

    !repeat 2: [ mybar1 ].

render: mysnippet
```

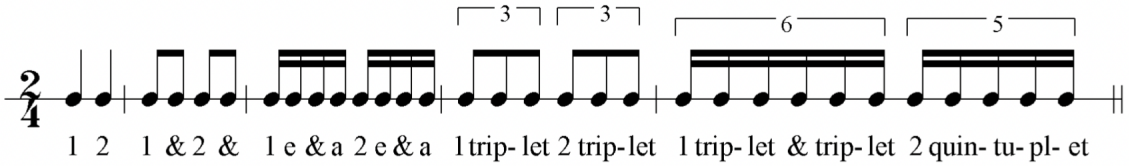


4 Language Concepts

There are two core concepts a user should understand in order to write programs in this language. First, how to notate different parts of the drum-set. Below is an example of a basic drum kit and the corresponding abbreviations we will use to write programs.

```
CC = Crash cymbal
HH = Hi-hat
RD = Ride cymbal
SN = Snare drum
T1 = High tom
T2 = Low tom
FT = Floor tom
BD = Bass drum
```

There are also different techniques to play these, such as playing the bell of ride (rd[b]), open hi-hats (o), slightly open hi-hats (˜), closed hi-hats (hh by default or x), foot hi-hats (_), ghost notes (g), flams (f), rimshots (r), accented strokes (a) etc.
Secondly, a user should be familiar with the "1 e and a" counting system:



Now knowing this, we can assign patterns to each part of the drum kit. For example, in 4/4 time, we can assign to SN (the snare drum) the pattern `1 e | 2 | | 4 a |`, where | just symbolises division of the beat (not necessarily if it is unambiguous).

5 Syntax

A pattern is made up of notes. Patterns can be assigned to drums in a bar. Bars can be combined to create a beat. Bars inside a beat can be repeated and modified inside a repeat instruction. Notes are primitives, which have values 1 e + a. Patterns have string names. Bars have string names. Beats have string names. Repeat instruction accepts an integer as the value of how many times to repeat a bar. It also accepts an optional an argument that represents which bar to modify. Properties to modify are specified inside curly brackets. Modifying properties is similar to assigned patterns to drums. Can also keep the pattern the same but change the drum from one to another by using the -> sign.

```
<num>            ::= x ∈ Z+
<string>         ::= any string
<varname>        ::= <string> | <num>

<time>           ::= time: <num>/<num>
<division>       ::= division: <num>/<num>

<title>          ::= title: <string>
<subtitle>       ::= subtitle: <string>

<count>          ::= <num> | e | + | a
<sep>            ::= |
<beat>           ::= <count>+ <sep>
<pattern>        ::= <beat>+

<pattern_expr>   ::= pattern<varname>: <pattern>

<drums>                  := cc | hh | rd | sn | t1 | t2 | ft | bd
<drum_pattern_var>       ::= <drums>: <varname>
<drum_pattern_notes>     ::= <drums>: <pattern>
<drum_pattern>           ::= [ <drum_pattern_var> | <drum_pattern_notes> ]
```

```
<bar_expr>              ::= bar <varname>: <drum_pattern>+

<repeat>                ::= !repeat <num>: [<varname>+].
<change_option>         ::= <num> | every <num>
<change>          ::= $change <change_option>: <varname> (<num>){<drum_pattern>+};
<snippet_data>          ::= <repeat> | <change>
<snippet_expr>          ::= Snippet <varname>: <snippet_data>+

<render>        ::= render: <varname>
```

6 Semantics

Syntax	Abstract Syntax	Type	Meaning
time: 4/4	Time of uint8 * uint8	Settings	Top number tells how many beats should be in one measure, and the bottom number tells what value of note should get the beat.
div: 1/16	Division of uint8 * uint8	Settings	The minimum value that notes can divide into
title: My Title	Title of string	Settings	Title of the piece
subtitle: My Subtitle	Subtitle of string	Settings	Subtitle of the piece
1 e + a |	Num of int | E | And | A | Sep	Note	Denotes the division of a note as defined by time signature (bottom number). Beats should be separated by `Sep`
sn	Drum of | CC | RD | HH | SN | T1 | T2 | BD	Drum	Denotes a drum
pattern varname: 1 | 2 e + a | 3 | 4 |	Pattern of PatternName * (Note list)	Pattern	a Pattern is a string denoting the name of the pattern after the keyword 'pattern' and a list of notes after the semicolon
hh: string	Drum * PatternName	DrumPatternVar	pattern variable assigned to a drum after the semicolon
hh: 1 2 3 4	Drum * (Note list)	DrumPatternNotes	a pattern of notes assigned to a drum after the semicolon
bar string: hh: [1 | 2 | 3 | 4 |]	Bar of BarName * (DrumPatternVar list * DrumPatternNotes list)	Bar	a Bar is a string denoting the name of the bar after the keyword 'bar' and a list of DrumPatternVar or DrumPatternNotes after the semicolon
!repeat ¡num¿: [varname1 varname2].	Repeat of int * (BarName list)	SnippetData	repeat instruction will repeat given bars ¡num¿ number of times.
@change ¡num¿: varname cc: [1 —] ;	RepeatChange of int * BarName * RepeatOption * (DrumPattern list)	SnippetData	change instruction behaves similar to repeat in the was that it will repeat one bar ¡num¿ number of times but it also accepts change instructions inside curly brackets, which are drum-¿pattern assignments.
Snippet mysnippet: ...	Snippet of SnippetName * (SnippetData list)	Snippet	Snippet creates multiple measure long pieces by combining repeat and change instructions.

i. A primitive value is a note defined by a counting value such as `1`, `e`, `+`, `a`.

ii. Values `1`, `e`, `+`, `a` are combined to create a `pattern`:

```
pattern: <count>+
```

A `bar` can be created by applying various `patterns` to different parts of the drum-set (`hh`, `sn`, `bd`, etc).

```
bar <string>:
    <drum>: <pattern>
```

And a `Snippet` can be created by combining and/or repeating multiple `bars`.

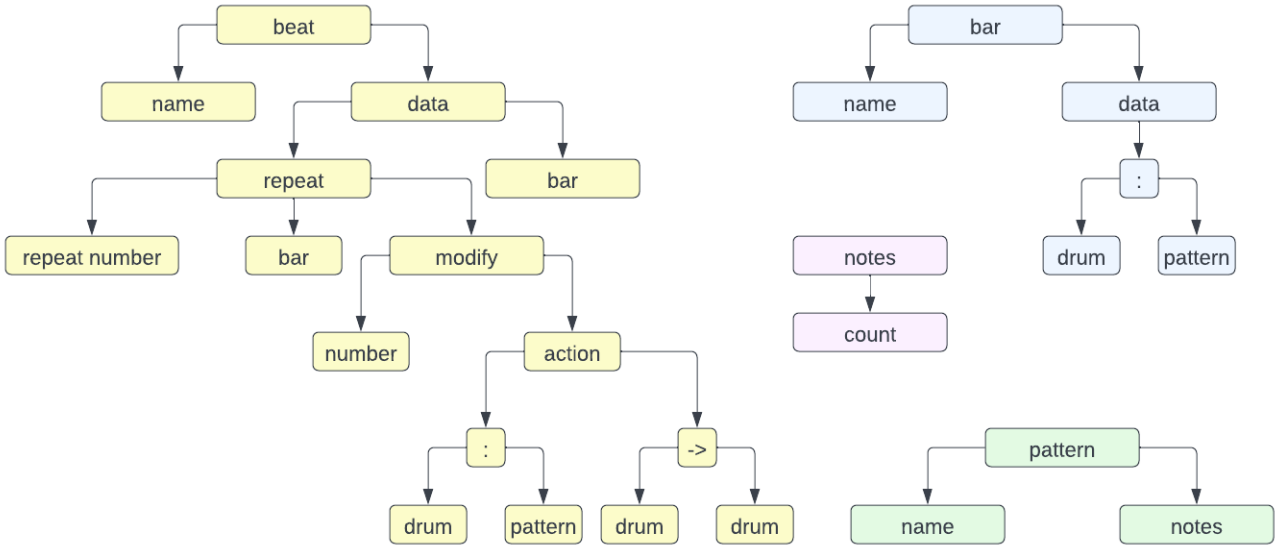
```
Snippet <string>:
    <bar>
```

`Bars` can be repeated by "`repeat <num> <bar>`", and the repeated `bars` can be modified inside curly brackets, where `<bar_num>` represents the bar number(s) to be modified:

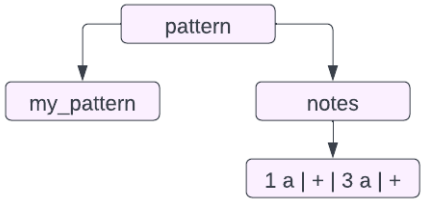
```
Snippet <string>:
    repeat <num> <bar> (<bar_num>) {
        <drum>: <pattern>
    }
```

Where `<bar_num>` represents the bar number(s) to be modified.

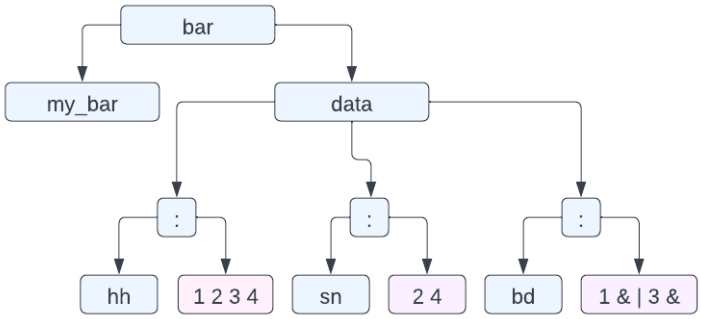
iii. Diagram representation of my program:



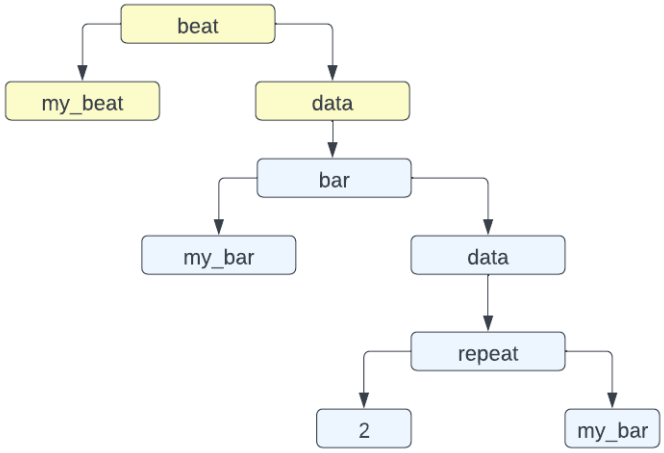
iv. AST for example 1:



AST for example 2:



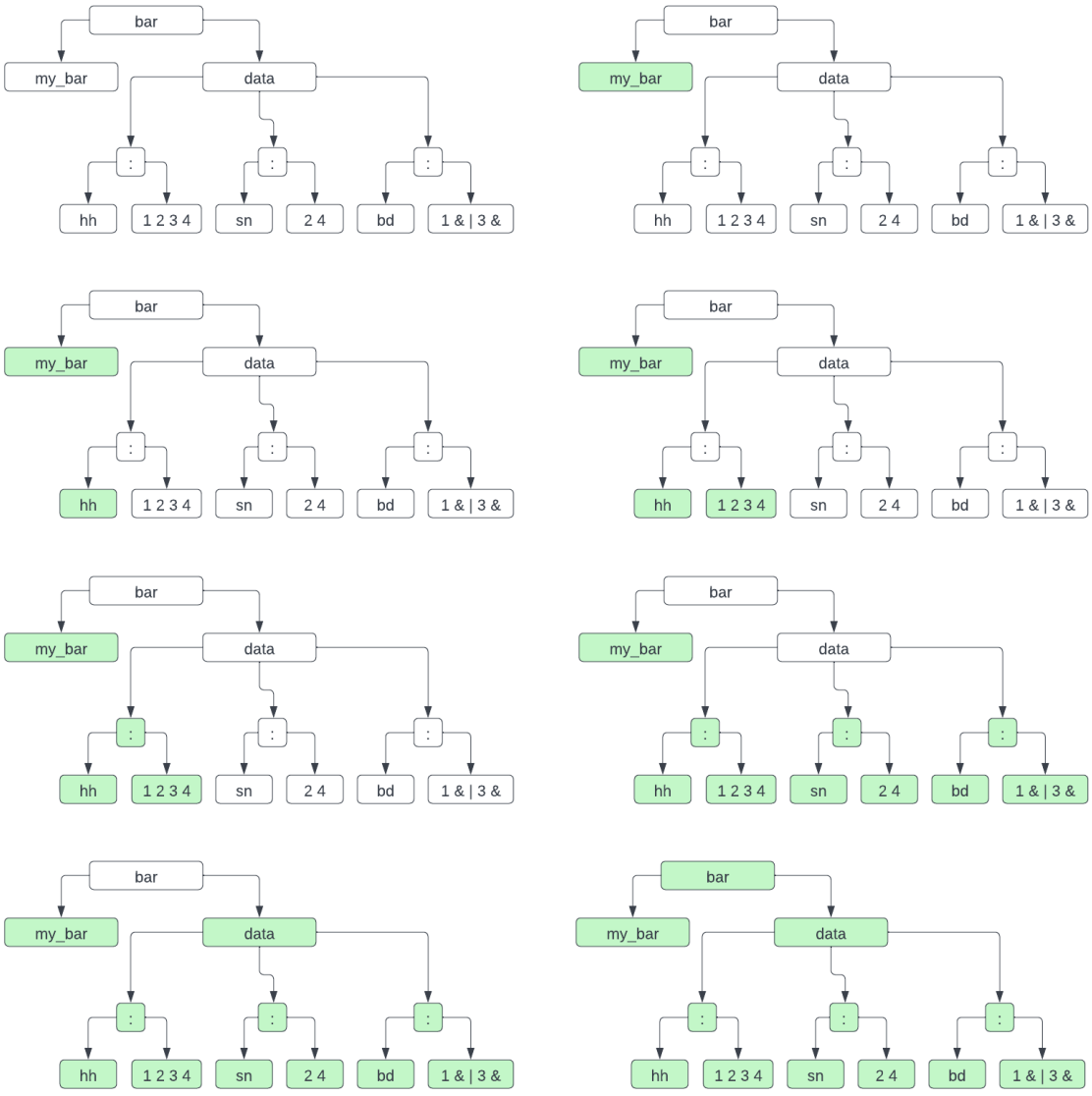
AST for example 3: (note that I've omitted the implementation of "my bar" as it is similar to example 2)



v. A. The programs do not read input.

B. The output is a pdf file.

C. Evalution of the program illustrated by example 2. First we evaluate the left branch, which is name of the bar. The we evaluate the right branch which contains actions as children. Each action has a drum name as the left node and a pattern as the right node. We first evaluate the drum, and then the pattern and assign the pattern to the drum. We continue doing so until we evaluate every action.



7 Remaining Work

The only thing remaining now is figuring out how to display rests and the beaming of notes. The distance of notes evaluates and displayes correctly, but I just need to add the beams and rests as necessary.
In addition, for a stretch goal there's a webapp where users can edit and play the pieces they wrote.