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

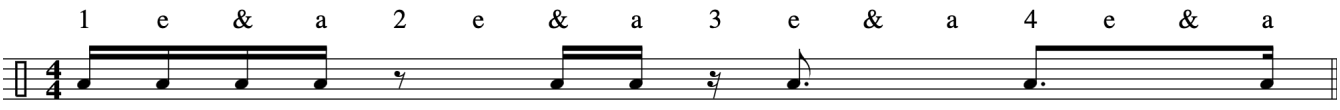
The most important design principle in this language is the fact that users can create and re-use patterns, bars, and beats that can be easily modified when writing the tabs. Perhaps in the future, users will be able to share the source code for each song written in this programming language on a platform 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.

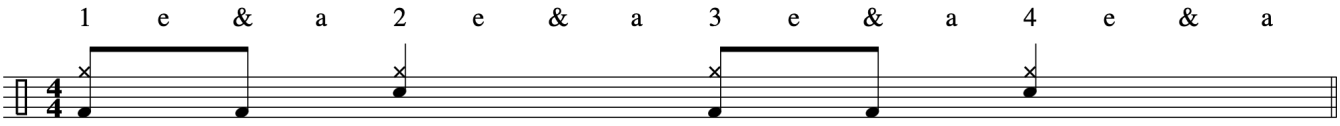
```
time 4/4
division 1/16

pattern my_pattern: 1 e + a | + a | e | 4 a |

render my_pattern
```



3.2 Example 2

Simple example of creating a one measure beat.

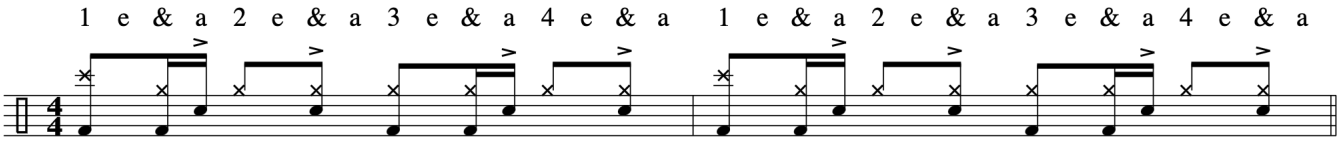
```
time 4/4
division 1/16

bar my_bar:
    hh: 1 2 3 4
    sn: 2 4
    bd: 1 & | 3 &

render my_bar
```



3.3 Example 3

Create a beat by repeating the pre-defined measure 2 times. There's an accent [a] on the snare drum.

```
time 4/4
division 1/16

bar my_bar:
    cc: 1
    hh: + 2 + 3 + 4 +
    sn[a]: a | + | a | + |
    bd: 1 & | 3 &

beat my_beat:
    repeat 2 my_bar

render my_beat
```

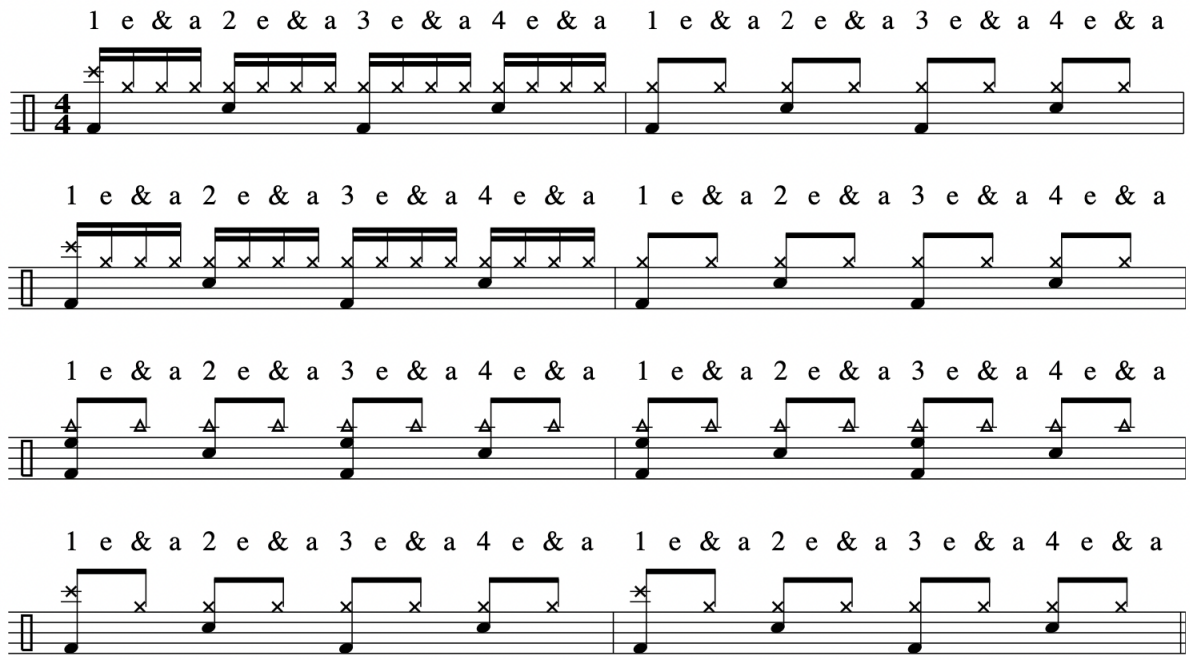


3.4 Example 4

We want the bass and snare drum patterns to stay constant, so we can create a new bar `main_bar` that we can re-use and alter later. We then can create a beat by repeating this measure 4 times and making every odd measure (1st and 3rd) have a different hi-hat pattern. We also changed the hi-hats to ride bell in the 5th and 6th measures, and added a crash cymbal on the first notes of the next two measures.

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

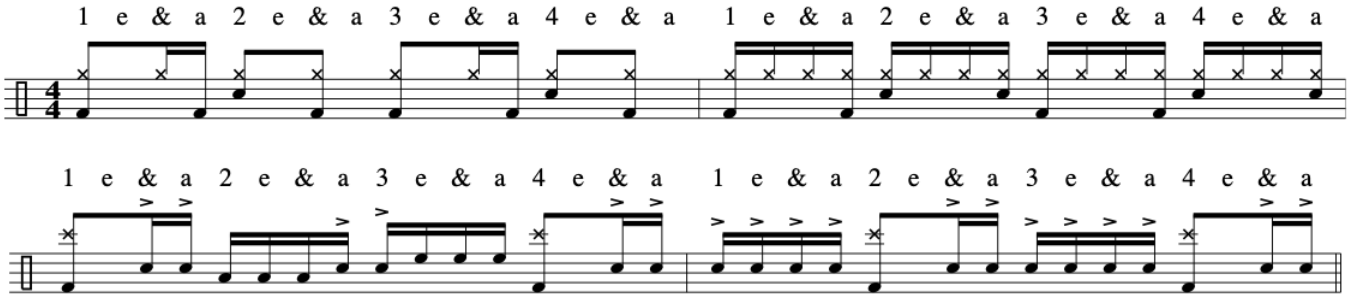
```
pattern 8th_note_pattern: 1 + | 2 + | 3 + | 4 + |

bar main_bar:
    hh: 8th_note_pattern
    sn: 2 4
    bd: 1 3

beat new_beat:
    repeat 4 main_bar (odd) {
        cc: 1
        hh: 1 e + a | 2 e + a | 3 e + a | 4 e + a |
    }
    repeat 2 main_bar {
        hh -> rdb
        t1: 1 3
    }
    repeat 2 main_bar {
        cc: 1
    }

render new_beat
```



3.5 Example 5

```
bpm 120
time 4/4
division 1/16

pattern new_bass_drum_pattern: 1 a | + | 3 a | +
```

```
bar new_bar_1:
  hh: 1 + | 2 + | 3 + | 4 +
  sn: 2 4
  bd: new_bass_drum_pattern

bar new_bar_2:
  hh: _ e + a
  sn: 2 a | 4 a
  bd: 1 a | 3 a

beat some_two_bar_fill:
  bar 1:
      cc: 1 4
      sn[a]: + a | a 3 | + a
      t1: 2 e +
      t2: || e + a
      bd: 1 4

  bar 2:
      cc: 2 4
      sn>: 1 e + a | + a | 3 e + a | + a
      bd: 2 4

beat main_part:
  new_bar_1
  new_bar_2
  some_two_bar_fill

render main_part
```



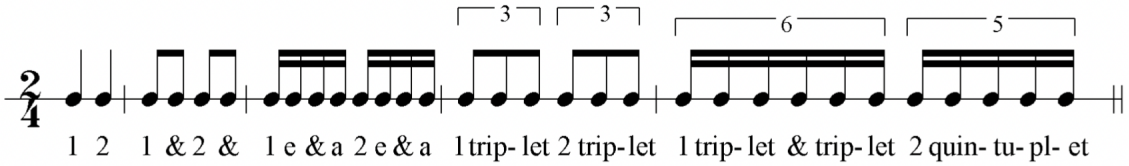
4 Language Concepts

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.

```
<ws>            ::= ␣ | ϵ

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

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

<count>         ::= <num> | e | + | a | |
<pattern>       ::= <notes>+

<hihat>         ::= hh
<snare>         ::= sn
<drums>         ::= cc | <hihat> | rd | rd[b] | <snare> | t1 | t2 | ft | bd
<variation>     ::= <hihat>[o] | <hihat>[˜] | <hihat>[x] | <hihat>[_] |
                    <snare>[a] | <snare>[f] | <snare>[r] | <snare>[g]

<pattern>       ::= pattern<ws><varname>:<ws><notes>

<drum_pattern_var>      ::= <drums> | <variation>:<ws><varname>
<drum_pattern_notes>    ::= <drums> | <variation>:<ws><notes>
<drum_pattern>          ::= <drum_pattern_var> | <drum_pattern_notes>

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

```
<beat>          ::= beat <varname><ws>:<bar>+
```

6 Semantics

Syntax	Abstract Syntax	Type	Meaning
$1 \in R, e, +, a$	Note of Num of int | E | And | A	Note	Denotes a note value
sn	Drum of HH | SN | BD	Drum	Denotes a drum
string	string	PatternName	variable name for the type Pattern
string	string	BarName	variable name for the type Bar
pattern string: 1 2 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

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 `pattern`s to different parts of the drum-set (`hh`, `sn`, `bd`, etc).

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

And a `beat` can be created by combining and/or repeating multiple `bar`s.

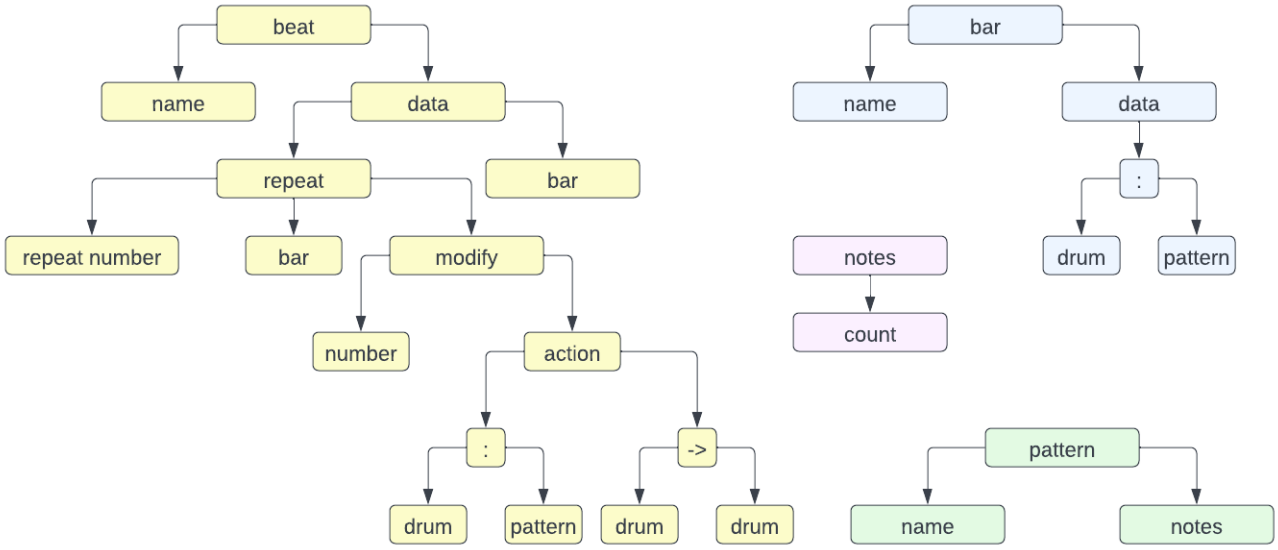
```
beat <string>:
    <bar>
```

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

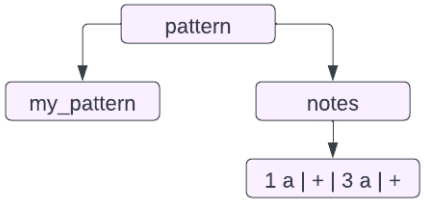
```
beat <string>:
    repeat <num> <bar> (<bar_num>) {
        <drum>: <pattern>
    }
```

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

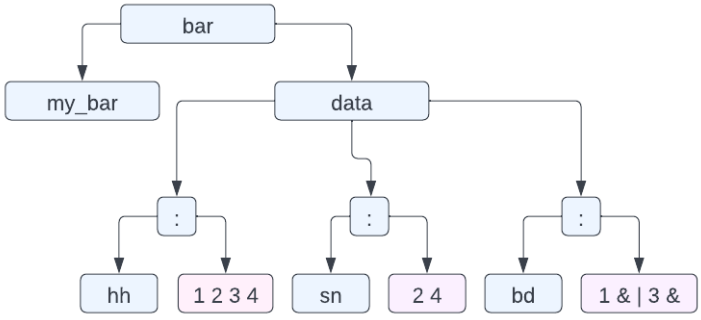
iii. Diagram representation of my program:



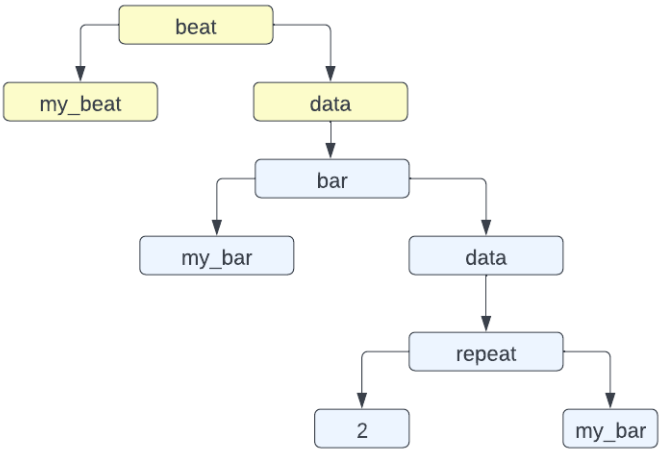
iv. AST for example 1:



AST for example 2:



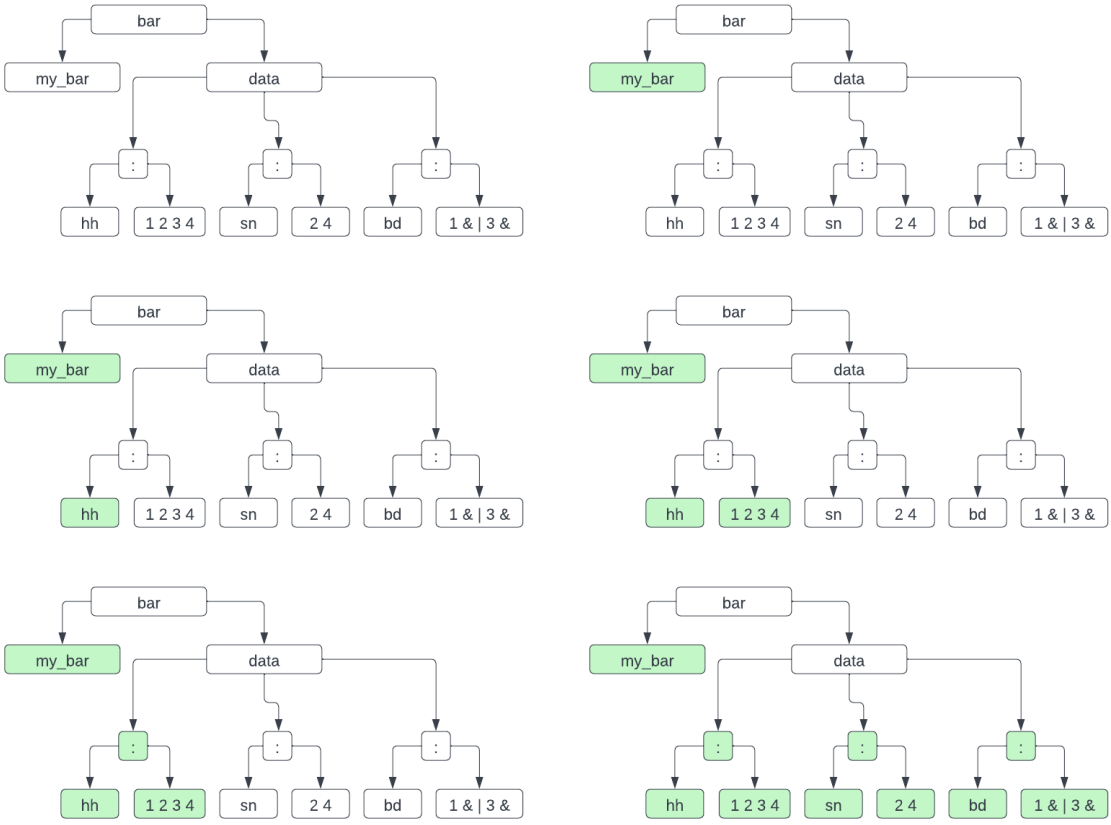
AST for example 3: (note that I've omitted the implementation of "my bar" as it is similar to example 2)

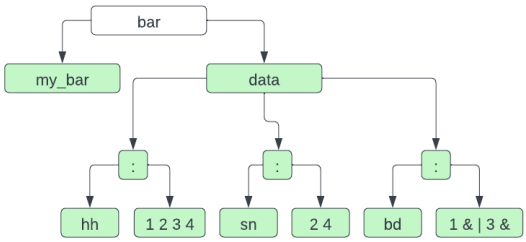


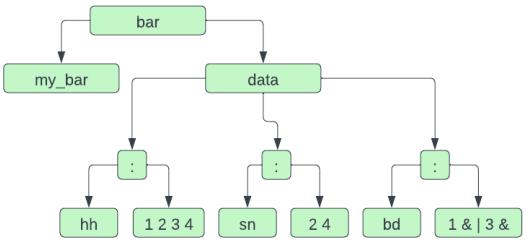
v. A. The programs do not read input.

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.




bar
my_bar
data
:
:
:
hh
1 2 3 4
sn
2 4
bd
1 & | 3 &


bar
my_bar
data
:
:
:
hh
1 2 3 4
sn
2 4
bd
1 & | 3 &