AlgoRhythm Documentation

Elias Devadoss, Himal Pandey

0.1 Introduction

Our language solves the problem of having to generate and record a song when you have an idea you want to implement. Instead, only the pieces to the song need to be given, and the language can output a fully realized song based on the program. While in the modern day we can have AI generate songs for us, there is still some difficulty communicating exactly what you want to a chatbot or similar service, as well as a lack of personal creativity in having a computer generate a song. On the flip side, being able to think of a song from nothing can be quite challenging and many do not know where to start.
This language serves as a jumping board for song ideas, where rhythm and melody can be mechanically reproduced and realized, without needing instruments on hand or physical skill. This specific language is also vital, as there are music production services where users can input exactly what they want to hear and receive an output. However, this language allows for a more loosely defined "song" to be created where a user only needs a relatively simple collection of chords, rhythm, and structure to be able to set the background for a song they are creating and test the waters. Finally, this language lowers the barriers for those who are not musically talented or inclined by making it relatively simple and easy to make a song, even for those with no experience or talent in the matter.

0.2 Design Principles

This language aims to be one that is simple and neat. The aesthetic will be one that aims for clarity and neatness. Due to the nature of how many specifics may have to be given, programs may be somewhat lengthy. However, they will be simple and easy to read. The idea is to have the language be one that follows the general principles of the language being easy to use and not a barrier for those who are not musically proficient. Therefore, the language will also be one that does not require learning much syntax or specifics, so that it is easy to implement.

0.3 Examples

i. Twinkle, Twinkle Little Star

```
<song> ::= 125 bpm 4/4 <melody> <beat> <chordList>
<melody> ::= C4 1 beats, C4 1 beats, G4 1 beats, G4 1 beats, A4 1 beats,
             A4 1 beats,G4 2 beats
<beat> ::= kick 1, snare 3, hi-hat 1 2 3 4
<chordList> ::= C4 E4 G2 2 beats, C4 E4 G2 2 beats, F4 A5 C5 2 beats,
                C4 E4 G2 2 beats
```

This will be executed via a command like `dotnet run "example-1.txt"`. The program will then be read and outputted to a MIDI file, which can be played by the user.
Output: Twinkle, twinkle little star with backing chords of C, C, F, C and a basic beat behind.

ii. 2:20

```
<song> ::= 65 bpm 4/4 <melody> <beat> <chordList>
<melody> ::= e 2.75 beats, A3 0.25 beats, C4 0.25 beats, D4 0.25 beats,
             F4 0.25 beats, A3 0.25 beats, C4 0.5 beats, D4 0.5 beats,
             e 0.5 beats, D3 1 beats, e 0.25 beats
<beat> ::= kick 1 1.5, snare 3, crash 1, ride 2 3 4
<chordList> ::= e
```

This will be executed via a command like `dotnet run "example-2.txt"`. The program will then be read and outputted to a MIDI file, which can be played by the user.
Output: The opening guitar riff of 2:20 by Colony House with a mock up of their drum beat and no backing chords.

iii. Ambient Sound

```
<song> ::= 60 bpm 4/4 <melody> <beat> <chordList>
<melody> ::= e 13 beats, D5 1 beats, Bb 1 beats, G 1 beats
<beat> ::= e
<chordList> ::= G4 Bb4 D5 A6 8 beats, D4 G4 A4 8 beats, A4 C4 E4 G4 B5 8 beats
```

This will be executed via a command like `dotnet run "example-3.txt"`. The program will then be read and outputted to a MIDI file, which can be played by the user.
Output: Creates an ambient background sound that can be looped and kept as a background noise or turned into the backing of a lofi song.

0.4 Language Concepts

A user needs to understand basic musical theory to write programs. They also need basic knowledge of how to write a computer program. If they do not have either of these, they still can likely write them, but there is a large chance they won't sound too good. In terms of primitives, users will need to know what different percussive sounds are, including both notes (which are percussive when played on a piano) and drum sounds.
Chords are a crucial combining form to understand. Chords are made up of a collection of three or more notes, and can be adjusted to add 2nds, 4ths, 7ths, 9ths, 11ths, and 13ths, depending on a user's wishes. A typical chord consists of a root, a third, and a fifth. A beat is created by repeating different percussive sounds at regular intervals on a meter. This serves as the heart of the inputted song. Finally melody and chords are combined by overlaying the tracks of notes on a meter and tempo given by the user.

0.5 Formal Syntax

```
<num> ::= positive integer
<song> ::= <tempo> <meter> <melody> <beat> <chordList>
<tempo> ::= <num> bpm
<meter> ::= <num> / <noteType>
<noteType> ::= 1 | 2 | 4 | 8 | 16 | 32
<melody> ::= <note>*
<note> ::= <pitch> <duration>
<duration> ::= <num> beats
<pitch> ::= <letter> <accidental> <octave> | e
<letter> ::= A | B | C | D | E | F | G
<accidental> ::= # | b | e
<octave> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
<beat> ::= <percuss>*
<percuss> ::= <sound> <num>+
<sound> ::= kick | snare | hi-hat | crash | ride | china | splash
<chordList> ::=  <chord>*
<chord> ::= <pitch> <pitch> <pitch>+ <duration>
```

0.6 Semantics

Our program is represented by the following components:

```
type song = int * meter * list note * list percuss * list chord
type meter = int * int
type note = pitch * int
type pitch = char * char * int
type percuss = string * list int
type chord = list pitch * int
```

Syntax: `song`
Abstract Syntax: `song` of `int * meter * list note * list percuss * list chord`

Type: `tuple`
Prec./Assoc.: n/a
Meaning: `song` is a combining form that holds all the necessary data for a song. It combines the tempo, meter, melody, beat, and chords.

Syntax: `meter`
Abstract Syntax: `meter` of `int * int`
Type: `tuple`
Prec./Assoc.: n/a
Meaning: `meter` is a combining form that tells the meter of a song, which is which note gets "the beat," and how many beats per measure

Syntax: `note`
Abstract Syntax: `note` of `pitch * int`
Type: `tuple`
Prec./Assoc.: n/a
Meaning: `note` is a combining form that gives a certain pitch and tells how long the pitch should be held out for.

Syntax: `pitch`
Abstract Syntax: `pitch` of `char * char * int`
Type: `tuple`
Prec./Assoc.: n/a
Meaning: `pitch` is a combining form that tells which note (C, D, E, etc.), any accidentals, and the octave of the given note. It can also be empty.

Syntax: `percuss`
Abstract Syntax: `percuss` of `string * list int`
Type: `tuple`
Prec./Assoc.: n/a
Meaning: `percuss` is a combining form that dictates a percussive sound, and which beats it should be played on. These form the heart of the beat.

Syntax: `chord`
Abstract Syntax: `chord` of `list pitch * int`
Type: `tuple`
Prec./Assoc.: n/a
Meaning: `chord` is a combining form that dictates the chord pitches, and duration.

0.7 Remaining Work

We still need to implement the parsed input and generates a MIDI file. This is the majority of the work that we have left. We also need functions such as parseMeter and parsePerc, so that the user can input the meter and percussive beats in a piece. Lastly, we need to tidy up some odd bits and pieces, such as parseLetter, which parses any letter, whereas letters that are not A-G should not be counted as successful parses.

The examples are included within the code folder.