AlgoRhythm

Elias Devadoss, Himal Pandey

PROJECT VIDEO:

```
https://youtu.be/bl2MUMV_Lr4
```

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 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 easy. 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 still maintain being 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.
A program will output two MIDI files, one for the chords and one for the melody. If either of those is empty or not specified, the MIDI file will be empty.

0.3 Examples

i. Twinkle, Twinkle Little Star

```
<song> ::= 150 bpm <melody> <beat> <chordList>
<melody> ::= Cn4 1, Cn4 1, Gn4 1, Gn4 1, An4 1,
             An4 1, Gn4 2,
<beat> ::=
<chordList> ::= Cn4 En4 Gn4 2, Cn4 En4 Gn4 2, Cn4 Fn4 An4 2, Cn4 En4 Gn2 2,
```

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. The actual implementation is below.

```
150 bpm
Cn4 1, Cn4 1, Gn4 1, Gn4 1, An4 1, An4 1, Gn4 2,

Cn4 En4 Gn4 2, Cn4 En4 Gn4 2, Cn4 Fn4 An4 2, Cn4 En4 Gn2 2,
```

Output: Twinkle, twinkle little star with backing chords of C, C, F, C.

ii. 2:20

```
<song> ::= 260 bpm <melody> <beat> <chordList>
<melody> ::= rr0 3, An3 1, Cn4 1, Dn4 1, Fn4 1, An3 1, Cn4 2, Dn4 2, rr0 2, Dn3 1, rr0 1,
<beat> ::=
<chordList> ::=
```

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. The actual implementation is below.

```
65 bpm
rr0 3, An3 1, Cn4 1, Dn4 1, Fn4 1, An3 1, Cn4 2, Dn4 2, rr0 2, Dn3 1, rr0 1,
```

Output: The opening guitar riff of 2:20 by Colony House with no backing chords.

iii. Ambient Sound

```
<song> ::= 60 bpm <melody> <beat> <chordList>
<melody> ::= rr0 13 beats, Dn5 1, Bb4 1, Gn4 1,
<beat> ::=
<chordList> ::= Gn4 Bb4 Dn5 An6 8, Dn4 Gn4 An4 8, An4 Cn4 En4 Gn4 Bn5 8,
```

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. The actual implentation is below.

```
60 bpm
rr0 13 beats, Dn5 1, Bb4 1, Gn4 1,

Gn4 Bb4 Dn5 An6 8, Dn4 Gn4 An4 8, An4 Cn4 En4 Gn4 Bn5 8,
```

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 sounds they wish to make, such as the sound of a piano key being played.
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. This serves as the heart of the inputted song. Finally melody and chords are combined by overlaying the tracks together.

0.5 Formal Syntax

```
<num> ::= positive integer
<song> ::= <tempo> <melody> <beat> <chordList>
<tempo> ::= <num> bpm
<melody> ::= <note>*
<note> ::= <pitch> <duration>
<duration> ::= <num> beats
<pitch> ::= <letter> <accidental> <octave> | e
<letter> ::= A | B | C | D | E | F | G | r
<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 * list note * list percuss * list chord
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 * 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: `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 Writing A Program

In order to write a program, the formalisms of our BNF grammar do not need to be followed directly. Programs should start with a tempo, of the form [n bpm], where n is the desired tempo. On the next line, a program should have the melody, consisting of notes with durations. A note has a pitch, an accidental, an octave, a space, a duration, and a comma, such as [Cn4 2,]. A user may input as many notes as they desire. Next is the beat, which for now simply consists of a newline character. The line after that is the chords, where a user puts multiple notes and then a duration, followed by a comma, such as [Cn4 En4 Gs4 4,]. This entire program should have four lines of code in total, with notes and chords separated by commas and having a trailing comma at the end of each note or chord as well. Please see the examples in the code folder for reference.

0.8 Running A Program

Run your program via dotnet run and with the name of your file in quotes. Example: `dotnet run "example-2.txt"`.

0.9 Remaining Work

We still need to implement setting the tempo of our outputted MIDI files. Tempo is parsed from the program, but not coded into the MIDI file. Additionally, we want to add percussion. This would require outputting another MIDI file, or implementing a new track within the MIDI file(s) we output.
We did not implement beat or tempo, as there is very little documentation for the program that we chose to implement, or other MIDI-coding softwares. In hindsight, it likely would have been decently easier to code the bits directly. However, there were some examples that we worked off of, and part of the time, the scant documentation website loaded. One of the biggest challenges we hope to overcome in the future is creating multiple tracks within a single MIDI file. We have our file set up in such a way as to accept multiple tracks, but when we created our files, only one track would play.
Finally, we want to eventually implement a function such that given a melody, the program can create chords and chord progressions to back this melody. This will allow a user to try out different sounds for their desired melody.