CS421 Final Project

Alexander Hansen

Friday 10th May, 2019

1 Language Overview

The language MusLang (like MusicLang, but slightly more creative) was created with for the purpose of creating music in an imperative programming style. Other projects such as Lilypond already exist, but they are for formatting and typesetting, not the actual creation of the music itself. The defining characteristic of MusLang is that the primitives are all musical scale degrees with a rhythmic duration. There are no numbers, strings, or other common data types. Whatever list of notes the main function returns is rendered to an SVG as sheet music. MusLang supports functions, for loops, variables, variable modification, collections, literals, and basic operations.

1.1 Primitive Literals

A primitive can only be one of two things: a list of notes or a note itself. A note takes the form of $nd.$ where n is the numeric scale degree, d is the duration (s, e, q, h, w for sixteenth, eighth, quarter, half, whole respectively), and . is an optional dot for a dotted rhythm. When the program is run, it is "anchored" to a key and the scale degrees are assigned based on that key. I find this method of writing music both convenient and musically intuitive. To express a dotted quarter note C in C major, the appropriate MusLang primitive would be $1q.$, as C is the first scale degree in C major. To express a D whole note, the appropriate MusLang primitive would be $2w$.

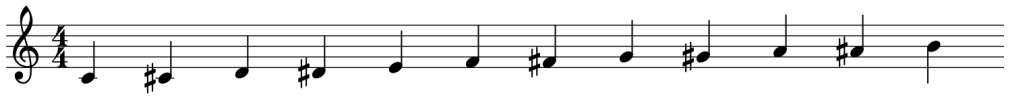
In order to express a sharp or a flat, you can prepend ^ or _ to the scale degree part of the primitive. This naming convention is what is used in ABC music notation and some other prominent software. A C sharp eighth note

can therefore be expressed as ^1e, and a D flat half note can be expressed as _2h.

2 Examples

2.1 Chromatic Scale

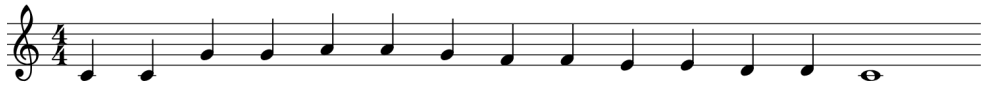
```
fn chromaticScale() {
      return <1q ^1q 2q ^2q 3q 4q ^4q 5q ^5q 6q ^6q 7q>
}
```



2.2 Twinkle Twinkle Little Star

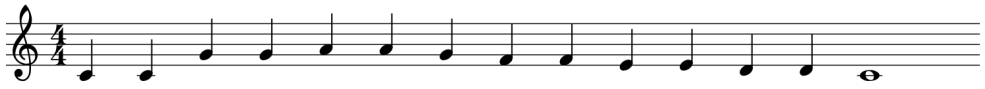
Here I used function application, variable assignment, and variable reassignment with an op to create the return variable, although it wasn't necessary.

```
fn main () {
        return twinkleTwinkle()
}
fn twinkleTwinkle() {
        let toReturn = <1q 1q 5q 5q 6q 6q 5q>
        let toReturn = op $toReturn + <4q 4q 3q 3q 2q 2q 1w>
        return $toReturn
}
```



2.3 Iteration

```
fn main () {
        let x = twinkleTwinkle()
        let toReturn = 1q
        for y in $x {
          let toReturn = op $toReturn + $y
        }
        return $toReturn
}
fn twinkleTwinkle() {
        let toReturn = <1q 5q 5q 6q 6q 5q>
        let toReturn = op $toReturn + <4q 4q 3q 3q 2q 2q 1w>
        return $toReturn
}
```



3 Implementation Details

3.1 Parser

For my parser, I used Parsimmon. It is "a monadic LL(infinity) parser combinator library" inspired by Parsec. Using this, I constructed my parser which parses the input and passes an object containing the parse result on to the interpreter.

3.2 Interpreter

My interpreter first evaluates the entire input without "going in" to any functions. This populates the global function and variable environments. MusLang keeps separate environments for the function declarations and variable declarations for the simplicity of typing, i.e. the function environment is a map from names to function bodies/args, and the variable environment is a map from names to expressions (which can be literals). Then, MusLang looks for a function called "main" in the function environment. If it doesn't find it, it returns an error. If it does, it evaluates the block assigned to the

"funcBody" of "main". There can be no arguments passed in to main, so it doesn't concern itself with matching arguments to parameters.

From here, the interpreter evaluates every line in the main function. Each line can be a for loop, function application, function declaration, operation expression, variable declaration, variable expression, return statement, or literal primitive.

3.3 Function Applications and Block Scoping

When evaluating a function application, MusLang evaluates a block. A block is a sequence of expressions contained within curly brackets ({}). When it evaluates the block, it inserts into its variable environment the arguments mapped to the parameters passed in (and throws an error if they do not match). This local block environment is exclusive to the block's execution context and therefore there are no conflicts among local variables declared within block scopes and global variables. Scoping works as it would be expected to in any other typical language.

3.4 For Loops

A for loop evaluates a block repeatedly, inserting the current value of the iterator into the for loop's variable environment. As previously explained, this ensures proper scoping and the iterator no longer exists after the for loop.

3.5 Operation Expressions

There is only one operation implemented right now, and it is addition (+). When adding, the return type is always a list of notes. A note plus a note is a list of those two notes. A list of notes plus a note returns the list of notes with that note prepended or appended, depending on order.

4 Limitations

There are quite a few limitations in this limited programming language. I have enjoyed working on this project and hope to expand MusLang in the future to handle these things.

4.1 Musical Limitations

The musical expressivity of MusLang is not great right now. There can be no ornaments or articulations. It is not really possible to express more complicated rhythmic or pitch ideas, like double dots, ties, half sharps, double sharps, etc. There are no dynamics. There can't be expression text attached anywhere to the document, nor can there be a title or description. Bar lines are not inserted according to the time signature, and everything is assumed to be in C major right now. You cannot express a chord, currently.

4.2 Technical Limitations

Due to weaknesses in my parser, variable expressions must be prepended with a dollar sign ($) and operation expressions must be prepended with the keyword "op". There are no booleans implemented right now, which means no comparisons and no conditional expressions. Parse errors and error reporting in general leaves something to be desired. You may need to check the console to see additional error output.