6.005 Project One Preliminary Design

TA: Leonid Grinberg

Team :

Vladimir Bok : vbok@mit.edu
George Pantazis: geopant@mit.edu
Henry Nassif: hnassif@mit.edu

Outline:

The high level organization of our project is the following:
The abc string is fed into the Lexer, which returns a sequence of tokens to the parser. The Parser handles the tokens and creates an Abstract Data Structure of Musical Objects that can be of one of the following types:

Note, Chord, Voice, MusicPiece.

Note, Chord, Voice and MusicPiece all implement MusicInterface.
MusicPiece is created by the play() method inside the Main Class. We will also use a MusicVisitor for which the MusicPiece will have the accept() method. The MusicVisitor will visit a MusicPiece and return a list<AugmentedMIDI> ready to be fed into the sequence player. Each AugmentedMIDI will hold its own start, duration and pitch. Important note: Tuplet is handled simply as a number of notes in succession. There is no special class for it.

Lexer

The Lexer splits a given abc string into header and body. To avoid any confusion between header and body of the abc string, the lexer will make the distinction early on and store each one in a different field, tokenizing each one according to their own grammar. When it comes to Token differentiation, will also differentiate between HeadTokens and BodyTokens. Even though Tokens with similar text/Symbol can be in both the body and the header, we would be able to distinguish them by creating instances of different classes (HeadTokens and BodyTokens).
After separating header from body, we will create tokens and assign them specific types by matching them with specific expressions (See Grammar). We will differentiate between header and body using Regular Expression (regex) matching which will subsequently be used to differentiate between Token types within the two separate fields.

We intend to make use of the following methods in our implementation:

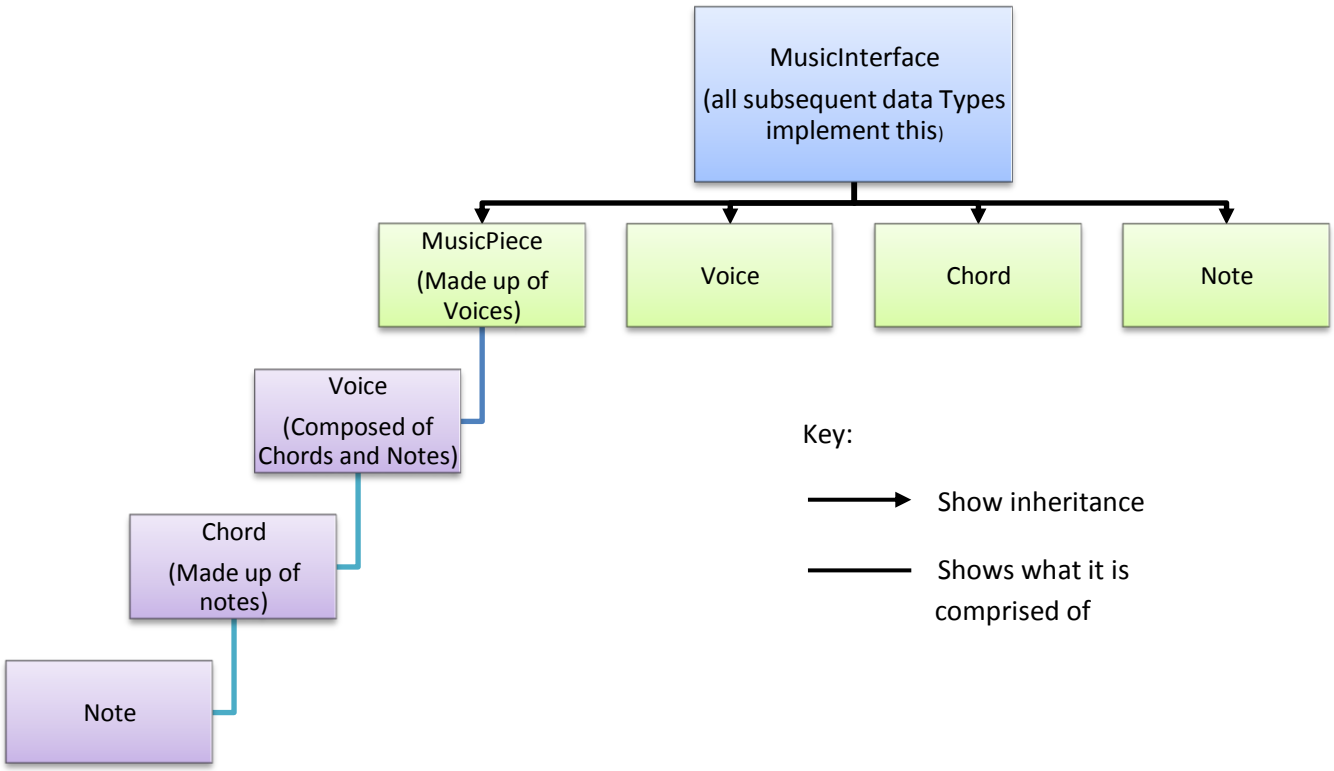
Lexer.Lexer():
The lexer's constructor will use given specifications to determine the start and end of the different parts of the input. The main Key for the distinction will be the string "K:" that is not part of a comment (ie: not preceded by "%") to determine the end of the header and start of the body.

Lexer.getNextHeadToken():
Method to iterate through the header and return HeadTokens. The "header" field string will be predetermined in the constructor. This method only operates on the "header" string.

Lexer.getNextBodyToken():
Method to iterate through the body and return BodyTokens. The "body" field string will be predetermined in the constructor. This method only operates on the "body" string.

Lexer.goBack():
Method used to handle repeats. Will move the lexer back to the start of a repeat segment. This method is called by the parser, whenever a token indicating the end of repeat segment is encountered by the parser. Due to the fact that certain tokens may now need to be ignored when moving a second time through a section, we will make us of a Delete() method.

Lexer.Delete():
This method is also called by the parser whenever some repeated tokens need to be discarded after the first repetition. Works in conjunction with goBack() so as to handle repeats which require ignoring a part of a piece of music when reading through a repeated section.



Parser:

The parser will return an abstracted version of the input: A musicalPiece which is a list of Voices where each voice will have a list of Chords, where each Chord will have a list of notes.

Method:

The Parser will apply the modifiers that precede the note, such as length, pitch and key to the note itself. We will have a modify() method that will associate each note with its modifiers and return a new note with the applied changes. When called on each node, the modify() method will also take into account the "sharp" and "flat" mentioned in the abcString head, which will be inherent modifiers.

Duration Field:

The duration of a note will be represented by a ratio of two integers in its simplest form. We will abstract the duration into two integers, one representing the denominator, and the other one representing the numerator, both of which would be private fields in the MultFactor class.

Interface MusicInterface:

accept(MusicalVisitor) //Accepts the visitor
getLength(Musical Obj) //Returns the length of the musical obj and uses dynamic dispatch to call the right getLength() function for each object (Song, Voice, Chord, Note)

class Note implements MusicInterface:

Private Pitch pitch;
Private MultFactor ratio

class Chord implements MusicInterface:

Private List<Note>
Private MultFactor duration

class Voice implements MusicInterface

Private List<Chord>
Private MultFactor duration

The visitor MusicVisitor will visit the different MusicPiece objects and recursively create a List<AugmentedMIDI> by appending created AugmentedMIDI Notes.

Testing:

We aim to test the following components of our system:

1. Lexer
2. Parser
3. Lexer and Parser together
4. Entire System

We have selected these components as this is where the majority of the work is done in our system. Moreover we expect most bugs to originate either in the Lexer or the Parser or in the interface between the two. More specifically:

1. **Lexer:** Here we will test that we can tokenize a variety of different inputs correctly. We will aim to have tests to explore each of the individual possible inputs, such as Notes, Chords, Tuplets, and multiple Voices. Moreover we will include tests that incorporate sequences of the above (notes, CHords, etc) and importantly tests for Repeats (this is not critical as much in the Lexer as is for the Parser).

2. **Parser:** The tests here are to ensure that the Parser correctly builds the AST to represent the Piece by properly identifying the tokens and applying all necessary Accidentals to individual Notes. As in the Lexer, we again aim to implement a full range of tests to explore a variety of possible inputs. An important test will be to examine whether Repeats are handled correctly as are multiple Voices.

3. **Lexer&Parser**: Here we test whether the interface between the two components functions correctly. Specifically, we want to ensure that one of the two does not interfere with the function of the other. Moreover this will allow us to test the ability of our System to handle Repeats.

4. **Entire System**: This will be the final test suite we implement and will be designed to ensure that inputs correctly propagate through our entire system and produce the correct outputs. This will allow us to test the overall functionality of our system and identify any lingering bugs. Its main purpose though will not be identification but rather for checking for correctness, since identifying bugs once they have propagated through the entire system is not an easy task.

Finally, as we progress and identify further problem areas, it may be necessary to augment our test suites to include tests of other components.