ABC Parser Project: Design Description

Overview:

We represent an ABC parser for music files as a series of well-structured interfaces. When an ABC Music file is selected from the menu, it will be read and converted into a string, which will then be passed to the lexer to create tokens, which are then fed to the parser to walk the tree and create `Music` objects, which then provide the necessary information to create and play the MIDI file created.

Revision Summary:

The lexer was changed to read duplet, triplet, and quadruplet parenthesis and digit as one token (so no quintuplets and up are allowed). The parser was changed to include a line rule that groups measures with lyrics to make adding lyrics to particular measures easier. It also was changed to include duplet, triplet, and quadruplet rules, also to make it easier to find and modify the notes associated with those.

Main was modified to include a pop up window to select the song to play. This user interface (UI) is designed to help the user play particular songs. They are divided in three categories which is done for clarity. The file implements a simple JFrame UI which makes it easier for the user to play any valid .abc file. The instructions are simple:

- Copy the .abc file into any of the three *_abc folders (otherSongs, sample, or songs).
- Run main and click on the song (it gets uploaded automatically)

Abstract Syntax Tree

MusicSymbol interface represents a music symbol, which can be either a pitch, a rest or a chord. The objects are immutable. The equals, toString, and hashCode methods work recursively and individually different from each class extending Music. Read their documentation for full specs (interface package).

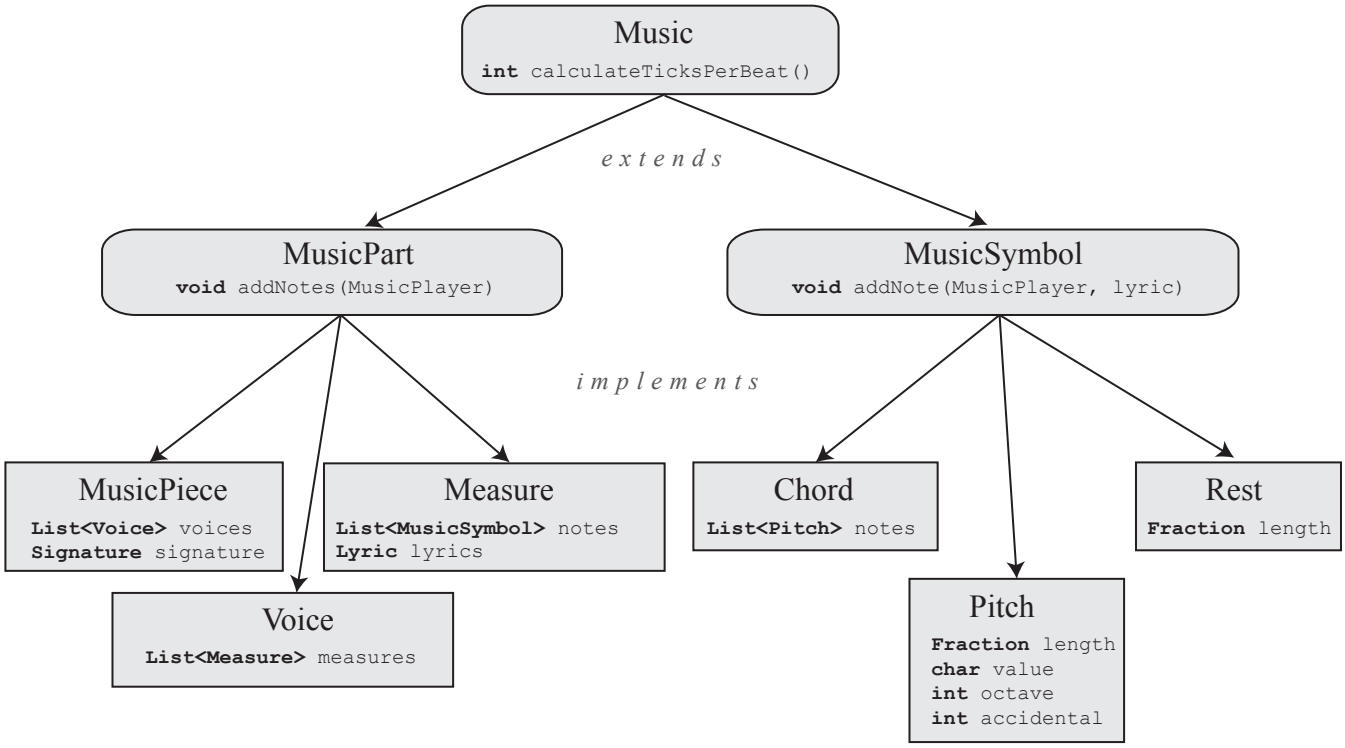
Representation MusicSymbol = Chord(*notes* : List<Pitch>) + Pitch(*value* : string, *octave* : int, *length* : int) + Rest(*length* : int)

MusicPart interface represents any music part. This could be a MusicPiece, a Measure, or a Voice. The objects are immutable. The equals, toString, and hashCode methods work recursively and individually different from each class extending Music. Read their documentation for full specs (interface package).

Representation MusicPart = MusicPiece(*signature* : Signature, *voices* : List<Voice>) + Measure(*notes* : List<MusicSymbol>, *lyrics* : Lyric) + Voice(*name* : String, *measures* : List¡Measure¿)

Dependency Diagram

Following is the diagram which underlies our abstract datatypes.



`Music` is an interface that will represent our ADT. All `Music` objects are immutable. It defines the following method

- `calculateTicksPerBeat` is an auxiliar method needed to determine the number of ticks per beat for the player such that any notelength can be represented as an integer number of ticks. The method is called recursively.

`MusicPart` is an interface that extends `Music` and represents either a `MusicPiece` or `Voice`. It defines a method

- `addNotes` which takes an instance of `MusicPlayer` (described later) and adds to it all notes and lyrics found in the current `MusicPart` . It is called recursively.

and is implemented by the following classes:

- `MusicPiece` represents a final piece of music. It has a `signature` attribute which contains all the header information and `List<Voice> voices` attribute which represents a list of different voices contained in this piece.
- `Voice` represents a single voice in a piece of music. It has a `List<Measure> measures` attribute which represents the sequence of measures the voice is made of.
- `Measure` represents a single measure in a voice. It has a `List<MusicSymbol>notes` attribute which represents the sequence of notes contained in the measure and a `List<Syllable> lyrics` attribute that represents the lyrics that accompain the measure.

`MusicSymbol` is another interface that extends `Music` and represents either a `Pitch`, a `Rest` or a `Chord`. It defines a method

- `addNote` which takes an instance of `MusicPlayer` and a lyric corresponding to the current symbol and modifies the player by adding notes and lyric to it.

and is implemented by classes:

- `Pitch` represents a single pitch. Its attribute `length` is represented by a fraction of the length of the default note. The attribute `value` is a pitch A,B,C,...,G from the middle octave, `octave` represents the offset from the middle octave and `accidental` is 1 for sharp and -2 for flat. Using these conviniences, a `Pitch` from out ADT can be easily converted to the `Pitch` object described in the `sound` package.

- `Rest` represents a single rest and has an attribute `length` represented by a fraction of the length of the default note.

- `Chord` represents a chord of several pitches and stores them in the `List<Pitch>` attribute.

These three basic music symbols let us implement any "musical expression" defined in the 6.005 subset. Any other structures like tuplets, triplets, repeats, etc. are converted to these basic music symbols during the ParseTree walk.

We also use an immutable class `Signature` inside MusicPiece that has all header information stored in its attributes and a mutable class `Lyric` which represents a list of lyrics used for each measure.

Another important class is a mutable `MusicPlayer` which has two attributes

- `player` that represents an instance of the `SequencePlayer`. It collects the notes and lyrics of the song.

- `ticksPerBeat` which represents the number of ticks per beat in the `player`

- `currentTick` which represents the current tick inside the player. It is used when the notes and lyrics are added consecutively to the player by the method `addNote` to keep track of the position where the notes need to be inserted. `addNote` method increments it according to the length of the note.

It has the following methods:

- `addNote (int note,Faction length)` that takes a converted Midi note and inserts it in the `player` at the `currentTick` position.

- `addLyric (String Lyric)` that takes a syllable and adds it to the `LyricListener` at the `currentTicks` position.

- `addTime Fraction length)` increases the `currentTick` by the length of the note.

- `resetTime()` resets the `currentTicks` to 0. This is used when starting a new voice.

- `play()` plays the notes and lyrics added.

Parsing

- **Lexer**: The lexer is designed such that all of the header lines (ie. title, composer, etc) are each lexed as one token, as are all lyric and comment lines. Notes are lexed together with their modifiers (accidentals, octaves, duration) as one token. Duplet, triplet, and quadruplet parenthesis and digits are lexed as a single token (apart from their notes). Chord brackets are lexed separately from their notes. Pipes, repeats, and end of measure symbols are lexed as their own tokens. All whitespace except for newlines (/n/r) is skipped.

- **Parser**: The parser has rules for the whole musical piece, which is then broken into the header rules and the actual musical body rules. Each header line has its own rule, with title, number and key being mandatory. The musical body consists of either lines, voices, and/or comments.

  Lines are measures followed by a newline, optionally followed by a lyric. Measures are note elements optionally surrounded by repeats or pipes, but must end with either a repeat, an end of measure symbol, or a newline.

  A note element is a note, rest, chord, duplet, triplet, or quadruplet. Duplets, triplets, and quadruplets are the particular header token followed by 2, 3, or 4 notes or chords, respectively. A chord is a number of notes or rests enclosed in square brackets. Note, rest, and lyric have their own rules as well, which are just their respective lexer tokens.

- **Errors**: Errors in parsing and lexing result in an exception being thrown. (reportErrorsAsExceptions is invoked on both parser and lexer).

- **Listener**: While parsing the tree, we care about these events: exit note, exit rest, exit duplet, exit triplet, exit quadruplet, exit chord, enter measure, exit measure, enter line, exit line, exit lyric, enter voice, exit header, exit tune. Several stacks are present at once, for each kind of object.

  When exiting the tune, voices and signature will be added to a MusicPiece object, which is then added to the stack.

  When exiting a header, the information is extracted from the context, and defaults are added if necessary. A default voice is set if none are provided. The current voice is kept track of.

  When entering a voice, the current voice is switched.

  When entering a line, a new list of repeated ranges is created.

  When exiting a line, lyrics are matched with their notes in their measure, and padded as necessary. Measure objects are created and added to the stack. If the measure index matches an index in the repeated range, that measure is added again (to the top of the stack).

  When entering and exiting a measure, repeats/n-th repeat/end of line symbols are searched for and the repeated range list is updated accordingly.

  When exiting a lyric, the chunk of text will be sent to another lexer and parser, which will return a list of lists of syllables, which will be pushed to the stack.

When exiting a chord, the number of notes or rests inside is determined, then they are popped, and then added to a Chord object, which is then added to the stack (this object represents a list of notes, since they all start at the same time).

When exiting a duplet, triplet, or quadruplet, notes inside are popped, and their duration is modified accordingly, and the new notes are inserted back into stack.

When exiting a note or rest, we extract the needed information from the context and add a Pitch or Rest object to the stack.

- **Tests**:
  - Lexer: Test different inputs and make sure tokens are broken up correctly. Examples: Simple header only, extended header, header and body, lyrics, music that includes chords, tuplets, repeats, music that has comments, music with all different modifiers
  - Parser: Test different inputs and expect fully-formed MusicPiece objects to be returned. Examples: Just a Signature, Pitches, Rests, Chords, Lyrics, multiple Voices, repeated measures, all kinds of notes and modifiers
  - ADT tests: test the methods of our ADTs Examples: equals, toString, hashCode, calculateTicksPerBeat, multiplyDuration, multiplyDuration, getLength, addNote, addNotes, etc.
  - Play: Testing the whole system, making sure that a file will be read in correctly, then lexed, parsed, and played. Examples: pass in all of the sample_abc files to be played, they contain the varying inputs that we want (simple measures, measures with comments, multiple voices, repeated sections, chords, tuplets, rests).

Parsing Lyrics

General idea The lyrics parsing is designed to be simple and flexible and at the same time robust and capable of parsing everything deemed a good input.

- **Lexer**: We start at the bottom token, LINESPACE. New line tokens are skipped because we have no special action to take when these are encountered. WHITESPACE follows and would normally also be skipped but because of the special WHITESPACE HYPHEN token, we must keep it in (this is explained with more detail in the parser rules). PIPEs are also unique tokens as they are used to separate measures. Then we have all the different symbols used to create different kind of syllables. Most importantly to note is that EXTENDERS and STARS can be one or more of the same while HYPHENs, DOUBHYPHENs, and UNION_OPERs are composed of single characters. Finally is the WORD token which creates the most complex form of lyric possible, and is composed of all allowed, non-special characters in a lyric.

- **Parser**: These rules define the structures used by the parser: A well constructed lyric will be parsed as follows

  - lyric:
    - can have multiple measures and any amount of trailing whitespace at the end
  - measure:
    - can be an entire lyric if no PIPEs are found
    - can be empty or a cluster (the first word in the cluster determines the first token)
    - if it begins with a PIPE, it may have WHITESPACE*
    - if it ends with a PIPE, optional WHITESPACE* may be used to separate the last syllable
  - syllable:
    - can ONLY start with a WORD or a STARS
    - single syllable and single STARS can be considered a full cluster
    - no combination of WORD and STARS are allowed in the same syllable
    - WHITESPACE* is allowed at the end of every syllable
    - if starting with a WORD it must be followed by:
      - a HYPHEN andor EXTENDERS
      - a WHITESPACE and HYPHEN andor EXTENDERS
      - a DOUBHYPHEN
      - a UNION_OPER
      - EXTENDERS

- Anything outside of these rules will be considered an incorrectly inputed lyric and will throw an exceptionfail silently with an unexpected outcome

- **Errors**: Errors in parsing and lexing result in an exception being thrown. (reportErrorsAsExceptions is invoked on both parser and lexer).

- **Listener**: The lyrics listener is very simple as well. The lyrics object is composed of an ArrayList of ArrayLists of Strings, where each String is a "syllable". So, when entering a new Measure, we add an empty ArrayList to the main ArrayList. The only other event is exiting a Syllable. In this case, we consider every valid input and do a second "parsing" accordingly. This is actually the pretty-print that we implement for our frontend. It converts things like two-word syllables joined by " " into two words joined by whitespaces.

  The end product is an ArrayList¡ArrayList¡String¿¿, not null, and processed enough to use a literal toString() method to print to the user's console.

- **Tests**:
  - Lexer:
    * Tests the LyricsLexer, ensures tokens are what they should be. Tests all kinds of lyrics.
    * Testing space:
      · We assume valid inputs. We create tests which lex one token at a time. Once every token is tested for, we introduce multiple tokens in the same input stream. A stream with all tokens is the upperbound while a stream with one token is the lowerbound.
    * Testing stragey:
      · Test each token individually. Test multiple tokens in complex combinations.