6.005 Project One Design

Project Title: An ABC Karaoke Player

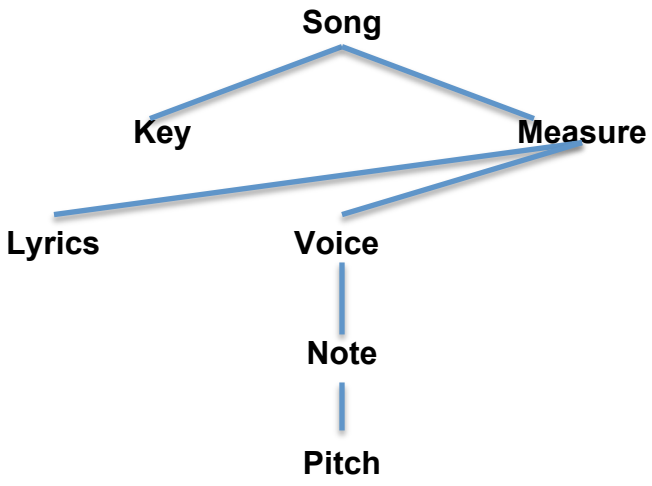
Team Members: Austin Freel, Josh Haimson, Christopher Ford

General Project Outline

File → Lexer → Parser → Listener → MIDI Generator → MIDI Player/Lyric Display

The .abc file will get fed to the lexer by way of a character stream. The lexer will output a token stream that will get fed to the parser. From this token stream, the parser will generate our abstract syntax tree. Our listener will walk along this abstract syntax tree, generating an instance of our abstract data type. This instance will get fed into the MIDI generator which will output a SequencePlayer and a LyricListener to the MIDI player, which will play the song.

Abstract Data Type Outline



Every song is a series of measures played in succession. Each song has a key which will add accidentals (sharps/flats) to every measure in the song. Additionally, each measure has it's own altered key as once a sharp or flat is introduced in the measure, it holds for all of the same notes in the measure. Each measure can be broken down into two more subunits: voices and lyrics. Each voice within the measure is a series of notes and each note stores a series of pitches and the note's duration.

Abstract Data Type Basic Specs

- All
  - toString – takes the object and returns a string representation of it
  - equals – returns whether or not two objects are structurally equal
  - hashCode - returns a hashCode for a given object
- All Mutable Objects
  - clone – clones the object to prevent representation exposure
- Song
  - Mutable
  - addMeasures – adds measures to the song
  - toSequence – takes the song and outputs a SequencePlayer
  - playSong – plays the SequencePlayer by way of MIDI player
-
- Key
  - Mutable
  - alterKey – changes the mappings of notes to pitches
  - getPitch – returns the correct pitch for a specific note within a key
- Measure
  - Mutable
  - addVoice – adds a voice to the measure
  - addLyric – adds lyrics to the measure
  - doRepeat – returns the number of the starting number of the repeat and marks it as repeated
  - alternateEnding – returns the measure of the alternate ending of a repeat sequence
  - modifyKey – modifies the key for the particular measure (adding an accidental)
- Lyric
  - Immutable
  - getDuration – returns the duration for the lyric in ticks
- Voice
  - Mutable
  - addNote – adds a note to the voice
  - getNotes – returns the notes within the voice
- Note
  - Immutable
  - getPitches – returns the pitches for the given note
  - getDuration – returns the duration of the note in ticks
  -
- Pitch
  - Mutable
  - accidentalTranspose – transposes a pitch up or down a number of semitones
  - octaveTranspose – transposes a pitch up or down an octave
  - transpose – transposing a note up or down a number of tones

- difference – the difference between two notes in semitones
- toMidiNote – returns a MIDI note of the pitch
- lessThan – returns whether or not a pitch is less than another pitch
- checkRep – checks the representation invariant

Grammar

At this point, we've elected to not modify the provided grammar.

ANTLTR Strategy

By way of developing and implementing a proper grammar, we plan on using ANTLR to generate for us a lexer to lex through the input file to generate proper tokens for later use. Similarly, we plan on utilizing ANTLR to develop for us a parser, using the properly developed parsing rules we have designed.

Abstract Syntax Tree Conversion

To transform our abstract syntax tree into a format that we can cleanly play using SequencePlayer, we will be writing a Listener similar to that from the second problem set. As noted above, our Listener will walk along our abstract syntax tree to generate an instance of our abstract data type. Again, as detailed above, this instance will be fed to the MIDI generator which will output to the MIDI Player to play the piece.

Testing Strategy

We will be implementing a "test first" programming style. We will write tests before writing code and will write tests incrementally as we go from tackling one project to tackling the next. Similar to in the second problem set, we plan on testing each main block of our code (i.e. lexer, parser, etc.). We plan on testing our Lexer's ability to generate appropriate tokens, our parser's ability to generate correct abstract syntax trees, our listener's ability to walk along the abstract syntax tree and develop an instance of our abstract datatype, the MIDI generator's ability to generate a SequencePlayer/Lyric listener, and finally, the MIDI Player and Lyric Display's ability to properly play notes and display lyrics. To generate our tests, we will be partitioning our input space, and will use a full Cartesian product strategy to ensure maximum coverage. We will test for bad inputs, for critical input values, and will brainstorm ways that someone might try and break our code and then test for that. We will be systematic with our tests, will test early and often, and will automate the running of our tests.

- Lexer Tests
    - Tests for finding rests
    - Tests for finding notes
    - Tests for finding invalid inputs and how to deal with them
    - Tests for whitespace
- Parser Tests
    - Tests for different orderings of tokens
    - Tests for invalid token orderings
    - Tests for proper token allocation
    - Tests for invalid inputs and how to deal with them
- Listener Tests
    - Tests to check for correct walking pattern
    - Tests to check for proper generation of abstract data types that build up to make a song(notes, measures, etc.)
    - Tests to check for proper generation of final instance of abstract data type (song)
    - Tests for invalid inputs and how to deal with them
- MIDI Generator
    - Tests for proper generation of SequencePlayers given a variety of inputs
    - Tests for proper generation of LyricListener given a variety of inputs
    - Tests for invalid inputs and how to deal with them
- Tests for MIDI Player
    - Tests for different durations (upper and lower bounds)
    - Tests for different pitches (upper and lower bounds)
    - Tests for invalid inputs and how to deal with them