6.005 Project One Design

Project Title: An ABC Karaoke Player

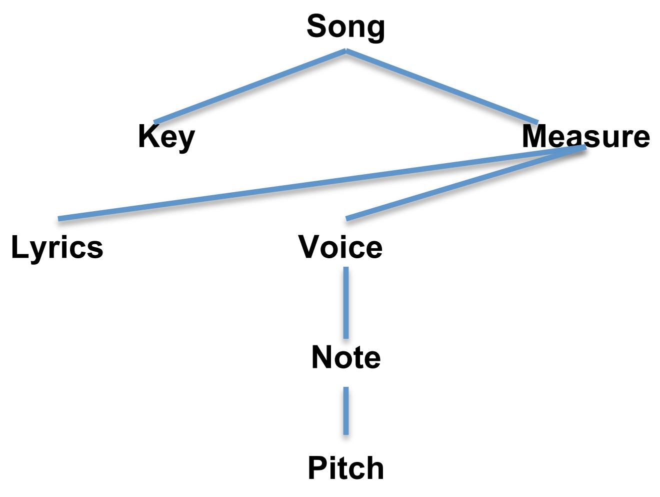
Team Members: Austin Freel, Josh Haimson, Christopher Ford

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. We've elected to break down measures into voices in order to account for cases in which multiple voices hit the same note that has been altered by an accidental earlier in the measure. This ensures that accidentals will be properly maintained through the measure.

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
  - Parameters:
    - Title: string representing title of the song
    - Composer: string representing the title of the song
    - Notelength: default duration of note
    - Meter: Sum of durations of all notes within a bar
    - ticksPerBeat: Amount of ticks for each beat usd by MIDI sequence player
    - Index: index number to determine play order if in playlist
    - Tempo: number of given-length notes per minute
    - Key: Key signature of piece
  - 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
  - Parameters:
    - keyName: a string representing the key of the piece
  - alterKey – changes the mappings of notes to pitches
  - getPitch – returns the correct pitch for a specific note within a key
- Measure
  - Mutable
  - Parameters:
    - ticksPerMeasure: amount of ticks in each measure
    - measureNumber: number of the measure
    - repeatStartMeasure: number where repeat starts
    - alternateEnding: measure to go to on alternate ending
    - hasAlternateEnding: whether or not an alternate ending exists
  - 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)

    - Note: to account for accidentals in a measure, for example, a measure that looks like A _A or C ^C, we have a private variable within each measure that keeps track of accidentals placed within a measure and will carry over those accidentals for repeated notes later in the measure.
- Lyric
    - Immutable
    - Parameters
        - Syllable: Syllable ot be displayed with each note
        - Duration: Number of ticks to display the syllable for
    - getDuration – returns the duration for the lyric in ticks
- Voice
    - Mutable
    - Parameters:
        - Name: name of the voice
        - ticksPerVoice: total amount of ticks in the measure
    - addNote – adds a note to the voice
    - getNotes – returns the notes within the voice
- Note
    - Immutable
    - Parameters
        - Pitch: Single pitch of note object to add to the Note
        - Pithces: A list of pitches within a chord to add to the Note
        - Duration: Number of ticks the note is played for
    - getPitches – returns the pitches for the given note
    - getDuration – returns the duration of the note in ticks
    -
- Pitch
    - Mutable
    - Parameters
        - C: character of the pitch to be created
    - 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

Using the provided grammar, we chose what we wanted to be parser and lexer rules, making slight modifications along the way. We've elected to make two grammars and thus two lexers and parsers for the project. One set will be used for the header of the file, the other for the body of the file. The header of the file contains information pertaining two the meter of the piece, the tempo of the piece, and the like. The body of the file contains everything else (notes, lyrics, etc.). Below are the lexer and parser rules for the grammars we've elected to use.

ABCMusicHeader.g4

```
/*
 * These are the lexical rules. They define the tokens used
by the lexer.
 */

X: 'X:' ;
T: 'T:' ;
C: 'C:' ;
L: 'L:' ;
M: 'M:' ;
Q: 'Q:' ;
V: 'V:' ;
K: 'K:' ;
W: 'w:' ;

BASE_NOTE : 'C' | 'D' | 'E' | 'F' | 'G' | 'A' | 'B'
        | 'c' | 'd' | 'e' | 'f' | 'g' | 'a' | 'b' ;

KEY_ACCIDENTAL : '#' | 'b' ;
MODE_MINOR : 'm' ;

NOTE_LENGTH_STRICT : DIGIT+ DIVIDE DIGIT+ ;

EQUALS : '=' ;

OCTAVE : [',]+ ;

DIVIDE : '/' ;
DIGIT : [0-9]+ ;
SPACE : ' ' ;
LINE_FEED : ('\r'? '\n') | '\r' ;
PERCENT : '%' ;

TEXT : ~[:\r\n]+ ;
```

```
COLON : ':' ;

METER_VARIANTS : 'C' | 'C|' ;
METER_FRACTION : DIGIT+ DIVIDE DIGIT+ ;
METER : METER_VARIANTS | METER_FRACTION ;

TEMPO : METER_FRACTION EQUALS DIGIT+ ;

KEY : KEY_NOTE MODE_MINOR? ;
KEY_NOTE : BASE_NOTE KEY_ACCIDENTAL? ;

COMMENT : PERCENT (TEXT | COLON)+ LINE_FEED ;
EOL : COMMENT | LINE_FEED ;

/*
 * These are the parser rules. They define the structures
used by the parser.
*/

abc_tune_header : abc_header EOF;

abc_header : field_number COMMENT* field_title
other_fields* field_key ;

field_number : X DIGIT+ EOL ;
field_title : T (TEXT | COLON)+ EOL ;
other_fields : field_composer | field_default_length |
field_meter | field_tempo | field_voice | COMMENT ;
field_composer : C (TEXT | COLON)+ EOL ;
field_default_length : L NOTE_LENGTH_STRICT EOL ;
field_meter : M METER EOL ;
field_tempo : Q TEMPO EOL ;
field_voice : V (TEXT | COLON)+ EOL ;
field_key : K KEY EOL ;
```

ABCMusicBody.g4

```
W: 'w:' ;
V: 'V:' ;
FIELD_VOICE : V (~[\r\n])+ EOL ;
```

```
// "^" is sharp, "_" is flat, and "=" is neutral
ACCIDENTAL : '^' | '^^' | '_' | '__' | '=' ;

BASE_NOTE : 'C' | 'D' | 'E' | 'F' | 'G' | 'A' | 'B'
        | 'c' | 'd' | 'e' | 'f' | 'g' | 'a' | 'b' ;

REST : 'z';

BAR_LINE : '|' | '||' | '[|' | '|]' | ':|' | '|:' ;
NTH_REPEAT : '[1' | '[2' ;

KEY_ACCIDENTAL : '#' | 'b' ;
MODE_MINOR : 'm' ;

NOTE_LENGTH_STRICT : DIGIT+ DIVIDE DIGIT+ ;

EQUALS : '=' ;

OCTAVE : [',]+ ;

OPEN_PAREN : '(' ;

OPEN_BRACK : '[' ;
CLOSED_BRACK : ']' ;

DIVIDE : '/' ;
DIGIT : [0-9]+ ;
SPACE : ' ' ;
LINE_FEED : ('\r'? '\n') | '\r' ;
PERCENT : '%' ;

LYRIC_TEXT : [a-zA-Z] ;

LYRIC_ELEMENTS : ' '+ | '_' | '-' | '*' | '~' | '\-' | '|'
;

METER_VARIANTS : 'C' | 'C|' ;
METER_FRACTION : DIGIT+ DIVIDE DIGIT+ ;
METER : METER_VARIANTS | METER_FRACTION ;
```

```
// note is either a pitch or a rest
NOTE : NOTE_OR_REST NOTE_LENGTH? ;
NOTE_OR_REST : PITCH | REST ;
PITCH : ACCIDENTAL? BASE_NOTE OCTAVE? ;

NOTE_LENGTH : (DIGIT+ DIVIDE DIGIT+) | (DIVIDE DIGIT+) |
(DIGIT+ DIVIDE) | DIGIT+ | DIVIDE ;

TEMPO : METER_FRACTION EQUALS DIGIT+ ;

KEY : KEY_NOTE MODE_MINOR? ;
KEY_NOTE : BASE_NOTE KEY_ACCIDENTAL? ;

COMMENT : PERCENT (~[\r\n])+ LINE_FEED ;
EOL : COMMENT | LINE_FEED ;

/*
 * These are the parser rules. They define the structures
used by the parser.
*/

abc_tune_body : abc_music EOF;

abc_music : abc_line+ ;
abc_line : element+ LINE_FEED (lyric LINE_FEED)? |
mid_tune_field | COMMENT ;

element : note_element | tuplet_element | BAR_LINE |
NTH_REPEAT | SPACE ;
note_element : NOTE | MULTI_NOTE ;

// tuplets
tuplet_element : tuplet_spec note_element+;
tuplet_spec : OPEN_PAREN DIGIT ;

// chords
MULTI_NOTE : OPEN_BRACK NOTE+ CLOSED_BRACK ;

// A voice field might reappear in the middle of a piece to
```

```
indicate the change of a voice
mid_tune_field : FIELD_VOICE ;

lyric : W lyrical_element* ;
lyrical_element : LYRIC_ELEMENTS | LYRIC_TEXT+ ;
```

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. We plan on using two lexers and two parsers. One lexer parser pair will be used for the header of the file (composer, key, etc.) and the other pair will be used to parse and lex through the body (notes, lyrics). We will also be using ANTLR to handle errors. With our correct grammar, we will use ANTLR's built in error handling to signal to the user when an .abc file does not adhere to the subset of ABC syntax we are handling.

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. We are planning on splitting the abc file into two parts, a header and a body, and we will have a Listener for each. Our strategy for our listeners is as follows:

Header Listener:

- Enter Header
- Enter each subpart of the header (Title, Key, etc.) and store the necessary values in an array
- Exit Header --> at this point we create a new 'Song' object using the values we obtained above (it is worth noting here that we will have an overloaded constructor for Song, so the absence of unnecessary fields is not a problem)

Body Listener:

- Enter Body
- Enter Line
- if Enter Comment --> ignore that line
- if Enter Voice --> store the name of the voice and starting measure number (we may return to a voice later; note that we do this because in our ADT, every Measure has list of Voices and a list of Lyrics. Storing

the starting measure of a voice allows us to synchronize voices with lyrics and voices with other voices)

- if Enter Element
    - Enter Measure --> create new Measure and add to Song
    - Enter Note OR Multinote OR Tuplet --> create new Note and add it to Voice
    - Enter Pitch --> new Pitch object, add to the Note
    - Exit Measure
    - (We may enter more Measures...until new line)
    - Exit Element
    - if Enter Lyric (note that there might not be lyrics, but if there are they will always be the line after Element) -->go through syllables and symbols and create new Lyric object, add this Lyric to Measure's lyric list

Rather than using a stack in our listener, we've elected to create an instance of Song and add to it as our listener walks our AST.

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
- ADT Tests
    - Testing equals(), hashCode(), toString()
    - Testing any as many methods of the class as possible