# 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 inputted, 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.
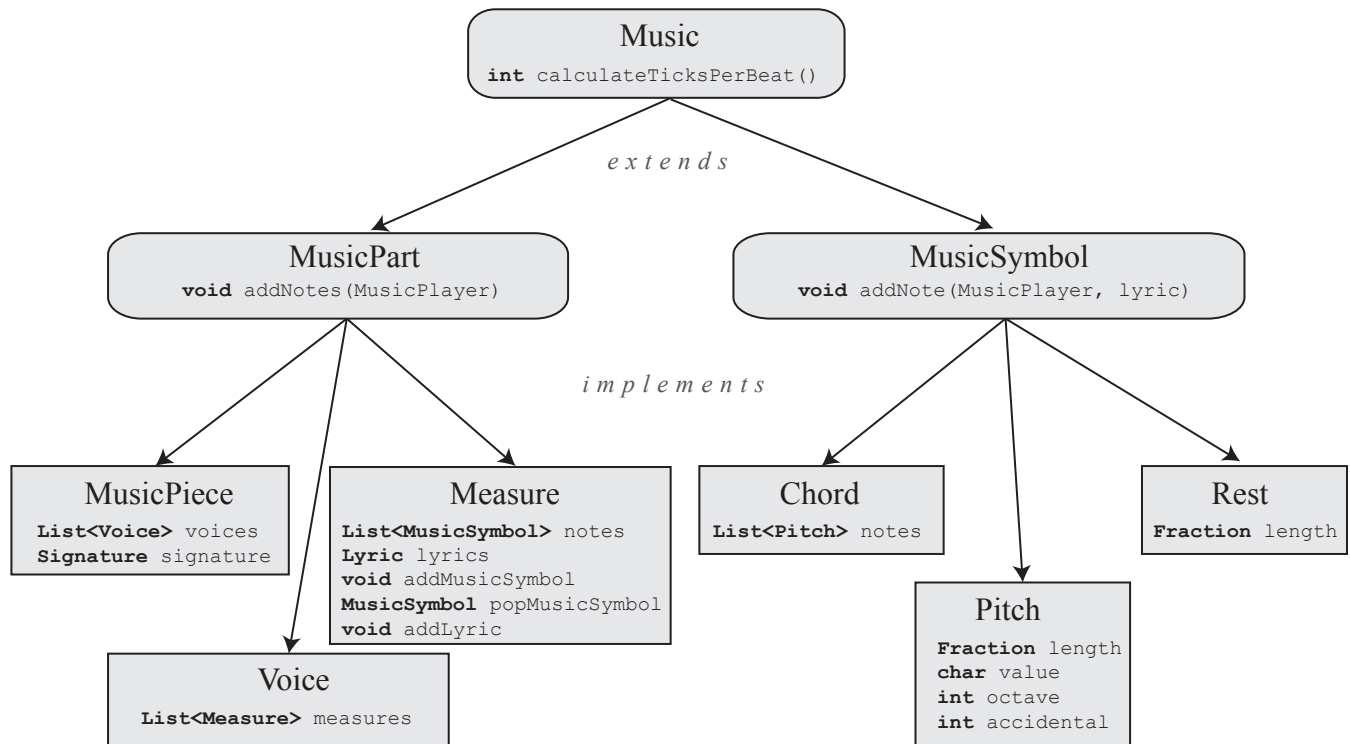
`Music` is an interface that will represent our ADT. It defines the following method

- `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`. All `MusicPart` objects are mutable. 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. It also has mutators `addMusicSymbol popMusicSymbol` and `addLyric`.

`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 llines. Notes are lexed together with their modifiers (accidentals, octaves, duration) as one token. Tuplet and chord brackets/parenthesis/digits 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 measures optionally followed by a lyric line or a voice or a comment. 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, or tuplet. Tuplets are a parenthesis followed by a digit and a number of notes or chords. A chord is a number of notes 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 only care about these events: exit note, enter measure, exit measure, exit rest, exit chord, exit tuplet, exit lyric, enter voice, exit header, exit music.

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

  When exiting a tuplet, the number of notes inside is determined, then they are popped, and their duration is modified accordingly, and the new notes are inserted back into stack.

  When exiting a chord, the number of notes 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 lyric, the chunk of text will be sent to another lexer and parser, which will return strings syllables, which will be added to the stack.

When entering and exiting a measure, repeats are searched for and the repeated measures are added to the voice. When exiting, the measure is created and added to the stack, gathering its notes and lyrics for their stacks.

When exiting music, voices and signature will be added to a MusicPiece object, which is then added 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 chord, tuplets, repeats, music that has comments

    - 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

    - ADT tests: test the methods of our ADTs Examples: equals, toString, hashCode, calculateTicksPerBeat, multiplyDuration, 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).

.g4 Grammar file

```
/*
 * These are the lexical rules. They define the tokens used by the lexer.
 *
 * All of the header lines and comments are individual tokens, then the notes and
 * rests are lexed together with their modifiers. Tuplet and chord (, [, and ]
 * symbols are lexed separately from their notes.
 * Repeats and pipes are lexed on their own.
 *
 */

WHITESPACE : [ \t]+ -> skip ;
DIGIT: [0-9]+;
NEWLINE: [\n\r];
INDEX : 'X' ' '* ':' ' '* [0-9]+ ' '* [\n\r]+;
TITLE : 'T' ' '* ':' ' '* [a-zA-Z0-9'.''  '','''!''#''&''('')'''?']+ ' '* [\n\r]+;
COMPOSER : 'C' ' '* ':' ' '* [a-zA-Z0-9'.''  ']+ ' '* [\n\r]+;
LENGTH : 'L' ' '* ':' ' '* [0-9]+'/'[0-9]+ ' '* [\n\r]+;
METER : 'M' ' '* ':' ' '* ('C' | 'C|' | [0-9]+'/'[0-9]+) ' '* [\n\r]+;
TEMPO : 'Q' ' '* ':' ' '* [0-9]+'/'[0-9]+ ' '* '=' ' '* [0-9]+ ' '* [\n\r]+;
```

```
VOICE : 'V' ' '* ':' ' '* [a-zA-Z0-9] ' '* [\n\r]+;
KEY : 'K' ' '* ':' ' '* [A-Ga-g]['#''b']?'m'? ' '* [\n\r]+;
LYRIC : 'w' ' '* ':' ('-' | ' ' | '|' | '\'' | '(' | ')' | '_' | '*' | '~' | ',' |
        '\-' | [a-zA-Z] | '.' | '!' | '?')+ ' '* [\n\r];
COMMENT : '%' ' '* [a-zA-Z0-9'.''!''?''-''_''*''~''\-''|'' ']* ' '* [\n\r]+;
NOTE :  ['^''^^''_''__''=']?[a-gA-G]['\''',']*([1-9]* '/' [1-9]+ | [1-9]+ '/'? |
        '/')?;
REST : 'z'([1-9]* '/' [1-9]+ | [1-9]+ '/'? | '/')?;
PAREN: '(';
PIPE: '|';
LBRAC: '[';
RBRAC: ']';
LREPEAT: '|:';
RREPEAT: ':|';
ONE_REPEAT : '[1';
TWO_REPEAT: '[2';
END_NOTES: '|]' | '||';


/*
 * These are the parser rules. They define the structures used by the parser.
 *
 * Each header field has its own rule. Notes, rests, tuplets, chords, and
 * measures have their own respective rules. Repeats have their own rules as
 * well, but to get the entire repeated measure, extract the token from measure.
 * Lyrics also have their own rule.
 *
 */
abc_tune : abc_header abc_music NEWLINE* EOF;
abc_header : field_number COMMENT* field_title other_fields* field_key;

field_number : INDEX;
field_title : TITLE;
other_fields : field_composer | field_default_length | field_meter | field_tempo |
               field_voice | COMMENT;
field_composer : COMPOSER;
field_default_length : LENGTH;
field_meter : METER;
field_tempo : TEMPO;
field_voice : VOICE;
field_key : KEY;

abc_music : (NEWLINE* measure+ NEWLINE* lyric? NEWLINE* | field_voice NEWLINE* |
            COMMENT)+;
measure : (l_repeat|PIPE)? note_element+ (END_NOTES|NEWLINE|r_repeat);

note_element : note | rest | chord | tuplet;
```

```
note: NOTE;
rest: REST;
tuplet : PAREN DIGIT (note|chord)+;
chord : LBRAC note+ RBRAC;
lyric: LYRIC;

l_repeat: LREPEAT | ONE_REPEAT | TWO_REPEAT;
r_repeat: RREPEAT;
```