ABC Media Player

PART 1: THE AST

Our music player AST will be composed of four main classes, **Piece, Measure, Voice** and **Cord**. These four main classes will compose our abstract syntax tree. We also will have the interface **MusicalAtom** and the interface **Sequence**.

Interface Sequence

This is an interface we will implement with **Piece, Measure, Voice** and **Cord.**

Methods:

public ImmutablePair<int,int> getShortestLength(); // This will recursively search for the shortest length note in the AST and return a pair of (numerator, denominator)

public List<MusicalAtom> getSequence( ); //This will return a sequence of musical atoms that represent the Sequence.

@overrid public boolean equals(Object that); // The equals method. Two sequences will be equal if they have the same sequence of equal musical Atoms

@overide public int hashCode();

@overide public String toString();

Abstract Class MusicalAtom

Fields:

private Pitch pitch;
private ImmutablePair<Integer, Integer> lengthCoeff;

Methods:

@override public boolean equals();
@override public int hashCode();
@override public String toString()

public ImmutablePair getLength() // This will get the length of the note
public void setAccidental(map accidentals) // This sets the accidental. This will modify pitch as necessary by looking up values in the accidentals map.
public void setLyrics(String syllableLyric) // This attaches a lyric to the note. lyrics will be matched during the exitMeasure() method of the listener.

Piece

Piece := List<Measures>

Piece implements **Sequence**. Piece is an immutable container class will hold a list of Measures as one of its fields and the rest of the abc header attributes in its other fields. It is immutability in the sense that its fields do not change but the contents of its field, like the list of Bars, are allowed to change if the field is mutable. Each Piece contains the Measures of a particular voice. Immutability is an invariant of this class. Another invariant of this is that it contains a list of measures.

Fields:

```
private final List<Measures> measures
private final String title;
private final String composer;
private final ImmutablePair<int,int> meter;
private final ImmutablePair<int,int> defaultLength;
private final int Tempo;
private final int indexNumber;
private final Key keySig; //Key is an enum
```

Methods:

```
@override public boolean equals();
@override public int hashCode();
@override public  ImmutablePair<int,int> getShortestLength();  // see Sequence interface

Public Piece(String keySig, int indexNumber, String title) //
All field not set by constructor will have their own set() or get()methods.
public void addMeasure(Measure m) // This will take a measure and add it to the measures
list.
public Key getKeySig() //This get method will be used to get the keySig attibute
@override public List<MusicalAtom> getSequence( ); // see Sequence interface
```

Measure

Measure := List<Voice>

Measure implements **Sequence**. Measure is an immutable container class that will hold a list of voices that are contained in each measure. It is also immutable in the sense that its fields do not change but the contents of its fields ,like the list of Voices, are allowed to change. An invariant of Measure is that every voice in the voices list must have the same number of beats.

Fields:

```
private final List<Voice> voices;
```

Methods:

```
@override public  ImmutablePair<int,int> getShortestLength();  // see Sequence interface
@override public List<MusicalAtom> getSequence(); // see Sequence interface
@override public boolean equals();
@override public int hashCode();
```

public void addVoice(Voice Voice) // This will take a Voice and add it to the Voices list.

Voice

Voice := List<Chord>

Voice implements **Sequence.** Voice is an immutable container class that will hold a list of Chords similar to Piece and Measure. Each voice hold one bar of cords in a single voice.

Fields:

private final List<Chord> chords;
private final map accidentals; // This store a dictionary with note as key and it the associated accidental as the value

Methods:

@override public ImmutablePair<int,int> getShortestLength(); // see Sequence interface
@override public List<MusicalAtom> getSequence(); // see Sequence interface
@override public boolean equals();
@override public int hashCode();

public void addAccidental(String accidental); //this will add any encountered accidentals to the
accidentals dictionary
public void addChord(Chord chord) // This will take a chord and add it to the chords list. It will check the map to see if any accidentals need to be applied to the cord before applying them.

Chord

Chord := List<MusicalAtom>

Chord implements Sequence. Chord contains musical atoms that are played at the same starting time and have the same length. Musical atoms will represent either notes or rests.

Fields:

private final List<MusicalAtom> atoms;

Methods:

@override public ImmutablePair<int,int> getShortestLength(); // see Sequence interface
@override public List<MusicalAtom> getSequence(); // see Sequence interface
@override public boolean equals();
@override public int hashCode();

public void addAtom(MusicalAtom atom) // This will take a atom and add it to the atoms list.

Notes

Notes := String note+ int startTick + int length

Note implements the abstract class **MusicalAtom.** A note is made of a Pitch representing it's pitch,

and an ImmutablePair, representing the length coefficient. The notes class is mutable. The fields are allowed to modified by other parts of the AST.

Fields:

private Pitch pitch;
private ImmutablePair<Integer, Integer> lengthCoeff;

Methods:

@override public boolean equals();
@override public int hashCode();

public Note(Pitch pitch, ImmutablePair<Integer, Integer> length) // This will initialize the class with the parse note pitch with the length of length
public ImmutablePair getLength() // This will get the length of the note
public void setAccidental(map accidentals) // This sets the accidental. This will modify string as necessary.
public void setLyrics(String syllableLyric)// This attaches a lyric to the note. lyrics will be matched during the exitMeasure() method of the listener.
public MusicalAtom getSequence()// getSequence on a musical atom will simply return *this*.

Rest

Rest := String note+ int startTick + int length

A rest implements musical atom abstract class. A rest is almost the same as a note, the only difference is that in a Rest setAccidental() has no effect.

PART 2: PLAYING THE AST

To play the AST we will call the the getSequence() method of the Piece.This will give us a sequence of notes to play. Next we will call getShortestLength() on the Piece. This will give us the length of the shortest note. We will use this length to calculate the ticks of each note when we call addNote(). The atoms contain all the necessary attributes needed as parameters for the addnote() method of the sequence player. We will simply iterate through the list of atoms and call addnote() on each atom, using the Pitch atom.note as a parameter to the addnote method.

Key signatures and accidentals are a particularly tricky part of the project. We will apply key signatures and accidentals as we are building our AST, since that is when we are instantiating the Pitch object that will be held by Note. Every time we parse a note, we will look to see if it has an accidental. If so, we will apply that accidental. Otherwise, we will have a keySig Map<Pitch, Integer> for each key signature. If a Pitch is in keySig, then we will apply to the pitch the key signature, represented as an 'accidental' on the note.

PART 3: TESTING

For the AST, we will need to test each class that forms the AST. Most importantly we will need to test the getSequence() method in Piece, Measure ,Voice and Cord. This can be done by creating notes and rests and then building AST then calling getSequence() method. It should return the notes in the

right sequence. For this test we will partition our input space into a single bar, multiple bars, a rest, a note, a cord, a single voice, multiple voices, a mixture of rests notes and chords.

Another important part to test is the playing the AST process. After generating the AST with the same input space partitions we will call the listener that will add our notes to the sequencePlayer. We can then use the same strategy as in the warmups to test our played output.

Finally we will have to unit test for equality and hashcodes.

PART 4: CREATING THE AST

ANTLR will parse a the abc file using the grammar. Runtime exceptions will be thrown for incorrect files. To create the AST we will use depth first search of ANTLR concrete syntax tree to build our AST.

First, when we enter the header part of ANTLR's tree, we will instantiate Piece with the values in the header such as title , composer e.t.c

Next, we will create a stack for Chords, and a Stack for lyrics. Exiting a note will add that note to the stack. Likewise exiting a lyric will add that lyric to the lyrics list. When exiting a Voice, we pop all the Chords from the stack and set their lyrics to the corresponding lyric using the setLyrics() method. Note that in our implementation we will represent a single note as a Cord with notes list of length 1. Similarly tuplets will be a Chord with multiple notes. Then we will add all the Chord in the stack to voice. The voice will be instantiated when we enter Voice and any accidentals we encounter will be stored in the accidentals map.

There will be another stack for Voices and a stack for measures. Like wise, exiting voices adds them to the voices stack and exiting measures add them to the the measures stack. exiting a measure adds all the voices in the stack to the measure and finally exiting a Piece adds or measures in the stack to the Piece.