ABC Media Player

Shantanu Jain
Dalitso Banda
Luis Sanmiguel

PART 1: THE AST

The AST will be comprised of several major classes. **Piece**, **Voice**, **Measure**, **Chord**, represent musical sequences, so they implement the <<**Sequence**>> interface. Underlying them are <<**MusicalAtoms**>> – **Notes**, and **Rests**. The **KeySignature** will be represented by an Enum. The AST is a linear representation of the music.

<<**Sequence**>>
Represents a playable musical sequence.

```
public IntPair getShortestLength();

      /**
       * @return the maximum number of beats in this sequence
       */
public double getNumberOfBeats();

      /**
       * @return List<?> A list of Chords or a list of List<Chord> that represents
       *    the Sequence.
       */
public List<?> getSequence()

public int hashCode();

public String toString();
```

Abstract Class <<**MusicalAtom**>>

```
private IntPair lengthCoeff;
private String syllableLyric;

public IntPair getLength()

public void setLyrics(String syllableLyric)

public String getLyrics()

public MusicalAtom clone()

public String toString()
```

Piece implements <<Sequence>>

Piece represents the whole song. In addition to the methods in Sequence
Contains header fields:

String title
IntPair meter
IntPair defaultLength
int tempo
KeySignature keySig
IntPair shortestLength
Map<String, Voice> voiceMap

```
public int getTempo()
```

```
	/**
	 * Adds a measure to the end of the Piece.
	 * @param voiceString name of the voice that's associated with this measure.
	 *    Must not be equal to "RESERVED_DEFAULT_VOICE"
	 * @param m Measure to be appended. Must not be null. Must obey the meter
	 *    of the piece.
	 */
public void addMeasure(String voiceString, Measure m)
```

```
	/**
	 * Append a measure to this piece. Only one of the addMeasure()
	 * methods can be called over the lifetime of this Piece Object.
	 * @param m Measure to add to the Piece
	 */
public void addMeasure(Measure m)

public KeySignature getKeySig()
```

Voice implements <<Sequence>>

A Voice represents one voice. It contains a list of Measures.

```
	/**
	 * Adds a measure to this voice. The measure must have the same number of
	 * beats as other measures added to this voice.
	 * @param measure The measure to be appended.
	 */
public void addMeasure(Measure measure)
```

Measure implements <<Sequence>>

A Measure represents one measure. It contains a list of Chords.

```
	/**
	 * Adds a chord to this measure. The chord must have the same number of
	 * beats per measure as other chords. @cr how will we handle this?
	 * @param c The Chord to be added to the measure
	 */
	public void addChord(Chord c)
```

Chord implements <<Sequence>>

Chord represents a set of MusicalAtoms, ie Notes and Rests, to be played simultaneously in the same Voice.

```
      /**
       * Adds a MusicalAtom to the Chord, to be played simultaneously.
       * @param a MusicalAtom to be appended. Must not be null.
       */
public void addAtom(MusicalAtom a)

public void addLyrics(String lyric)
```

PART 2: PLAYING THE AST

At a high level, the Lexer/Parser will take in an *.abc file, and call the CSTListener. This listener will then construct the AST in parts, making use of utility classes to handle repeats, alternate endings, KeySignatures, and accidentals. This AST, represented as a single Piece instance, will be walked by a walkPiece(Piece) method, which will make use of the SequencePlayer class to schedule notes and lyrics.

The walkPiece() method will be the key logic here. A Piece ::= List<List<Chord>>, with each sub-list representing a voice. Each sub-list's elements represents a Chord. A Chord contains many MusicalAtoms to be scheduled at the same time. Each Voice will be scheduled in parallel, with each Voice's Chords sequentially scheduled.

There are several design challenges and considerations that led to this design.
Deciding ticksPerBeat
We realized that we needed to decide ticksPerBeat after the AST had been constructed, because we needed to know the shortest note length to set the ticksPerBeat. Thus we didn't want to place any information in terms of ticks when constructing the AST. That's why we have a getShortest() method.

Key Signatures
We wanted to apply KeySignatures as we were building the AST, since we know the Key Signature at the time of parsing the notes. We also knew we needed a representation for KeySignature, so we created an Enum for it. We then created a KeySignatureUtility class, which held a Map between KeySignature and another Map, between Pitches and the accidentalValue (as an int). Clients would simply use the getAdjustedPitch() method, with a KeySignature and Pitch as arguments, to get back a new Pitch adjusted by the octave.

Accidentals
We wanted to apply the accidentals at the time of constructing the AST, since we knew the measure associations at the time of parsing. We made sure to apply the accidental only to the same note at the same octave, for the duration of the measure, only to notes after the accidental. We implemented this in the CSTListener as we were instantiating Notes.

Repeats and alternate endings
Since we decided that the AST was going to represent a linear sequence, we needed a way to linearize the “musical roadmap.” We created the MusicalRoadmapUtility class, to which you addMeasure() and addDemarcation() in order they appear in the music. Single bar lines are ignored. Only repeat symbols, first and second ending symbols, and double bar lines are observed. Once the client is done adding these elements, he calls linearize() to be returned a List<Measure> representing a linear version of the song, which is then added to the AST. Each voice has it's own instance of MusicalRoadmapUtility, solving the problem of voices that appear intermittently throughout the song.

Association of lyrics with Chords
A Chord can contain lyrics. These lyrics, if there are any, are then scheduled at the same time as the Chord themselves. The challenge we had was associating the Lyrics to the appropriate MusicalAtom, as they appear on different lines. Since Chords are instantiated in the CSTListener, walking the tree later to associate the lyrics would be difficult. So within the CSTListener, we added in 2 queues, one for lyrics and one for Chords. When we exit the lyrics line, we associate the lyrics with the Chords and put them into the measure. This was one of the most tricky parts of our implementation.

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, Key Signature, Meter, etc.

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 Chord with a list of MusicalAtom to be scheduled simultaneously.

Tuplets will be split up into different chords. 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.

We will associate measures and voices when a voice is completed. While we are associating lyrics and chords, we will also be pushing to the stack bar lines and demarcations. When we run into a bar line, we will associate all chords on the second queue to the measure. Finally, when we call linearize(), we will associate voices with the Piece.