6.005 Project I Design

David Yamnitsky (nitsky), Michael Wu (mwu2015), Sumit Gogia (summit)

AST Classes

Out AST is a series of one-to-many relationships between the following classes. Piece has a one-to many relationship with Voice, Voice with Measure, Measure with Chord, and Chord with Note. With this design, no lower tier class is found without the higher tier classes. For example, this means that all Note objects will be found in Chord, even if the Chord has just one Note.

Piece

The root node of our AST is an object of class Piece, whose instance variables include all the header information necessary to display and play an ABC file, as well as a list of Voice objects.

Voice

The voice class represents all the measures that make up a voice in a piece. Its instance variables feature a voice name and a list of Measure objects.

Measure

The measure class contains a list of one or more Chord objects. In addition, it keeps track of a RepeatType, which is one of BEGIN, END, FIRST_ENDING, SECOND_ENDING, SECTION_END, and NONE, which represent all the functions a measure can have in traversing through a voice with repeats.

Chord

A Chord object contains a list of one or more Note objects.

Note:

A Note object contains in its instance variables:

- A pitch, any one of A-G or z, the character reserved for a rest
- An accidental, the number of semitones to transpose the note by
- An octave modifier, the number of octaves to transpose the note by
- A length, measured in default note lengths (the value specified in the header)

Lexer

Our lexer operates entirely using regular expressions. We have a one-to-one mapping between token types and a regular expression that finds instances of it in the input. We combine all these regular expressions into one that we call the master pattern finder that pulls up all valid tokens from the input, and then we go through our mapping to determine which

Grammar

PIECE := HEADER_FIELD+ VOICE* (VOICE? VOICE_MUSIC)+
VOICE_MUSIC := MEASURE*
MEASURE := (BAR_BEGIN_REPEAT | BAR | FIRST_ENDING)? (CHORD | NOTE | TUPLET)+ (BAR)?
TUPLET_MUSIC := TUPLET NOTE+

CHORD := CHORD_BEGIN NOTE+ CHORD_END
NOTE := NOTE_ACCIDENTAL* NOTE_NAME NOTE_OCTAVE_MODIFIER* NOTE_LENGTH?

HEADER_FIELD := [CKLMQTX]:.*\n
VOICE := V:.*\n
TUPLET := ([234]
NOTE_ACCIDENTAL := ^+|_+|=
NOTE_OCTAVE_MODIFIER := '|,
NOTE_NAME := [zA-Ga-g]
NOTE_LENGTH := [1-9]*/[1-9]*|[1-9]+
BAR_BEGIN_REPEAT := |:
BAR := [| or || or |] or |
FIRST_ENDING := [1
SECOND_ENDING := [2
CHORD_BEGIN := [
CHORD_END :=]

Parser

Our parser is implemented with one parser function for each of our AST classes.

The parseNote function takes a series of tokens and produces a Note object parsed from it. It applies both the key signature and accidentals passed to it. A time factor is also applied to the length. This is used to handle tuplets, as described below.

The parseChord function takes a series of tokens and calls parseNote for each group of tokens it determines to form a single note. It then returns a chord containing all these notes.

The parseMeasure method takes a series of tokens and segments it into tokens representing chords and calls parseChord on each set. The parseMeasure method also allocates a map of accidentals which it passes to parseChord and down to parseNote. When parseNote finds an accidental, it updates the accidental map passed to it. In this way, all notes in a measure share a common map of accidentals. When parseMeasure comes across a tuplet, it calls parseChord passing down a time factor, or multiplicative factor on the length of all notes in the tuplet.

The parseVoice method is the highest level method in our parser. It takes a series of tokens and segments them into measures. For each measure, it calls parseMeasure and assigns the appropriate value of RepeatType for the measure.

Before the parser deals with the body of the input, though, there is a parseHeaders method which fills the Piece object with appropriate header information and provides default values. Next, we have a method called mergeVoices which collects all tokens associated with each voice and places them in map in which the keys are the voices and the values are the list of tokens for each voice. Then, parseVoice is called for each voice.

Key Signature

We implemented key signature with a class that reads an XML file containing each major and minor key and the sharps or flats associated with it. Below is an example of the declaration of D major in the XML format we devised:

```
<keysignature>
        <key>D</key>
        <type>Sharp</type>
        <note>F</note>
        <note>C</note>
</keysignature>
```

This indicates that the key of D Major contains two accidentals, sharps on the F and C.

Playback with the Visitor Pattern

To do playback, we use two different visitor classes with our AST.

NoteLengthVisitor

The note length visitor traverses the AST and keeps track of the denominators of all notes it visits. Then, it computes the least common multiple of these values, producing the number of ticks per default note length for the Piece. This is done so that each Note in the Piece produces an integer number of ticks, meaning we take into account how finely every note subdivides a default note length. This is done because of the requirement in the java MIDI sequencer that all notes be scheduled at integer time steps, called ticks. The value the note length visitor returns is passed to the SequencePlayer in our Main file.

SequenceBuilderVisitor

The sequence builder visitor takes a SequencePlayer and a Piece, and traverses through all the voices, measures, chords, and notes in the Piece, adding their midi representation to the SequencePlayer. As we traverse through the chords in the voice, we increment a position value by the length of each chord it passes. For each voice, the position value is reset to 0, so that voices play concurrently. When visiting a chord, the length of a chord is determined to be the maximum length of any of its constituent notes. In this way, we are able to accommodate chords with notes of varying length, with the caveat that no new notes can be played over a chord. This design makes sense to us, because having one note of a chord held while a series a notes play beneath it seems more like multiple voices, though it may make some piano music more tedious to transcribe. In the visit method for Voice objects, we pass over measures. This is where we implement repeats. Our initial design had a component of the parser which duplicated all measures in a repeat, but we decided instead to mark measures with their repeat functions instead so that error handling in the parser would provide more correct value of measure numbers in errors. If the repeats caused measures to be duplicated, measure numbers would be higher than expected and could confuse the user. So when the SequenceBuilderVisitor visits a measure, it checks the value of its RepeatType, and traverse through the measures accordingly.

Main

We implemented a simple main function that takes a single command line argument, the path to an ABC file, and attempts to play it back.

Error Handling

We put a great deal of effort into error handling, providing an elegant printout displaying what the issue was and where it occurred.

In the Lexer, there is a step that applies the master pattern matcher to the input and sees if there are any characters that are not caught by it. If there are, all the characters that don't get matched, which must be invalid characters, are printed out so the user knows which characters to remove.

The Parser has much more extensive error handling. For all cases we could conceive of unexpected and uninterpretable token sequences we called one of two different error functions, either throwParserException or throwParserBodyException. The former is used for errors in the headers, such as invalid value for the tempo, key signature, etc. The latter is used for errors found in the body, in which it conveniently prints our the voice and measure number where the error occurred. All of these errors cause IllegalArgumentExceptions to be thrown, which are caught in our Main file where their error message is printed to standard out.