Revised Design Milestone

Eric Ruleman (eruleman), Nicholas Mizoguchi (nickmm), Victor Horta (vhorta)

1. Describe your strategy for using ANTLR to create your AST. You should also describe how you will handle errors in the input file.

Overview

Our strategy consists of building our AST from bottom-up using the ANTLR concrete syntax tree obtained with our GRAMMAR using ANTLR. We start building the primitive elements, such as chords (one or more notes) and rests, and then we use them to create measures, and sections. We also take into account that we have to keep track of which voice we are building, and also align the lyrics correctly with its corresponding note while running through the tree.

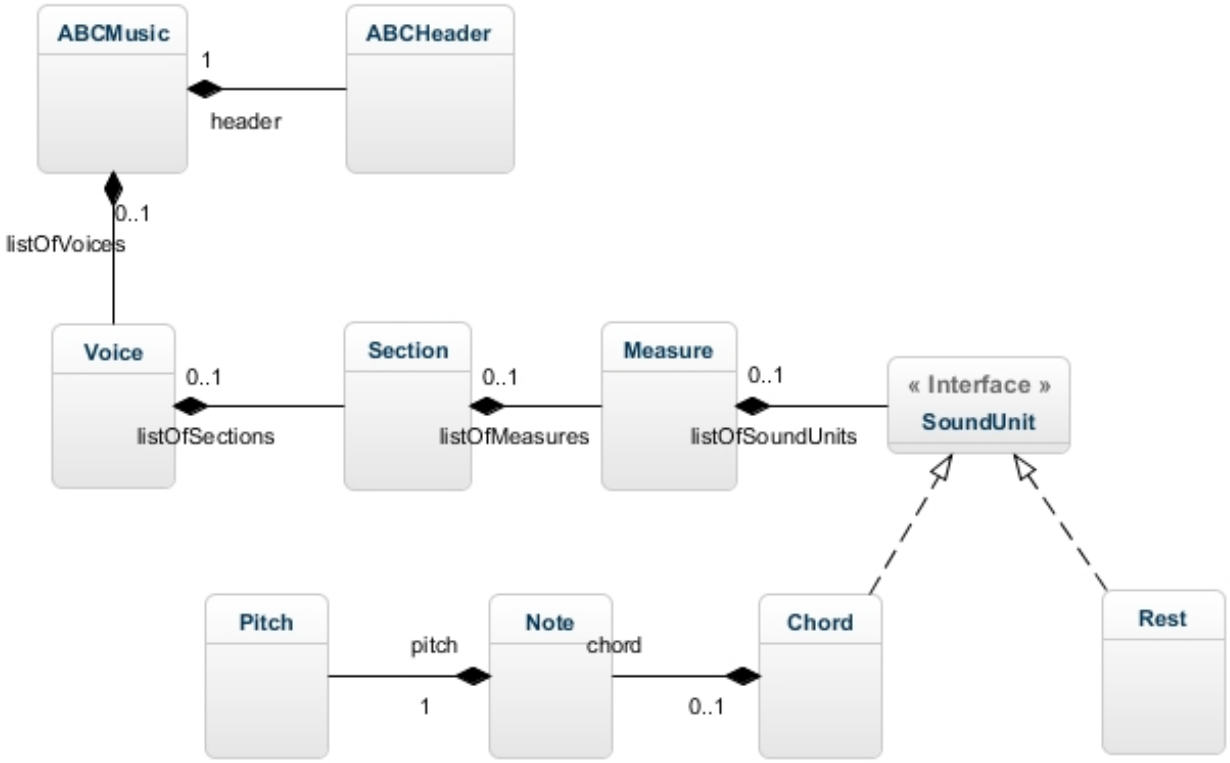
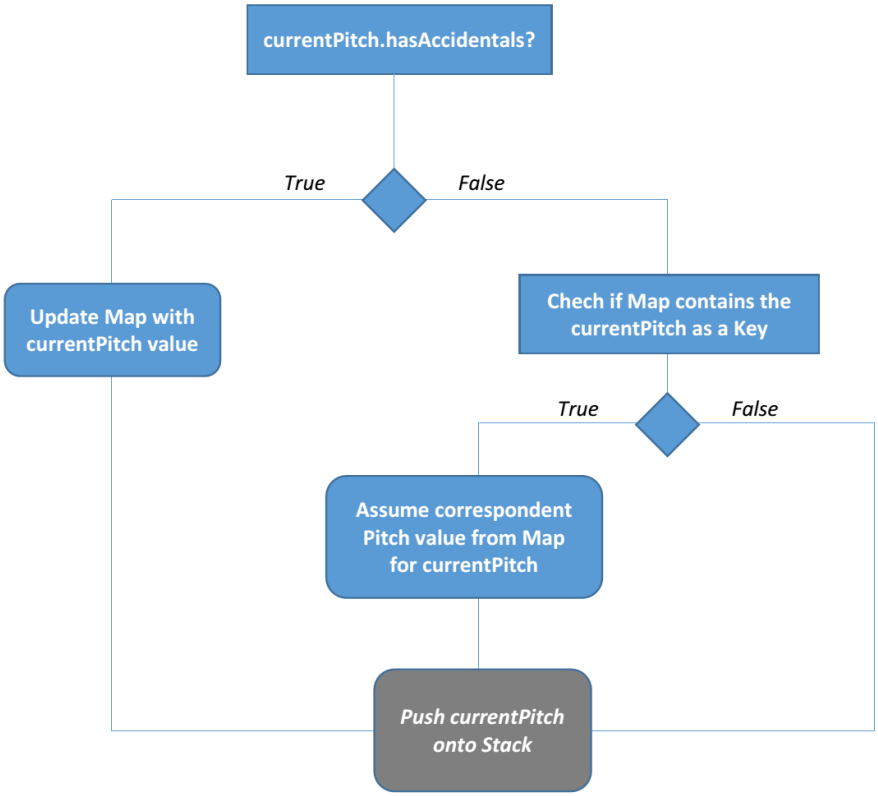


Figure 1 – Project 1 Class Diagram
Methods and Attributes are described at the end of the document.

Strategy

When we exit an accidental, we push the accidental as an Integer that represents that semitone shift. When we exit valid_note, we will push the node's value to a stack as a char. When we exit an octave, we push the node's value as an Integer that represents the octave shift. When we exit a pitch node, we instantiate a Pitch object by popping the corresponding number of children of the pitch node and use the information to instantiate a Pitch object (note that Pitch has been changed such that it contains the field hasAccidental to distinguish cases such as F and =F). But now we need to check if this pitch needs to be altered either because of the key signature or because an accidental came beforehand in the measure (accidentals carry for the entire measure). In order to do this, we will have a measureAccidental state. The measureAccidental is a Map<Pitch, Pitch> that by default represents the key signature. When we instantiate a Pitch object, we check the measureAccidental map in order to see if this new Pitch needs to be changed to another Pitch by checking if the key of the newly instantiated Pitch is in the map. If it is, then we delete the newly instantiated Pitch and instead instantiate a Pitch object that is equal to the value of the key in the measureAccidental and push that onto the stack. As we traverse the measure, we will update the measureAccidental state to include accidentals that we've seen so far in this measure. A new pitch with an accidental "overrides" the previous key value found in measureAccidental, which means that we will replace the value of the key with the new pitch. (Note: When we exit a barline node, we will "refresh" the measureAccidental such that it once again represents the key signature.).



.

Figure 2 – Pitch computation diagram

When we exit a rest node, we will push a 'z' char onto the stack (that represents a period of rest).

When we exit a note_length, we will instantiate a double and push that onto the stack.

When we exit a note node, there are two cases:

- ✓ Case 1: A note has two children which means that it has a note_length child. In this case, we will pop the double (that represents the note length) and an Object, that represents a Pitch or a period of rest. If the object is a Pitch, we instantiate a Note, and push it to the stack, and if it is a period of rest, we instantiate a Rest object and push it to the stack.
- ✓ Case 2 (Rest): A note has one child which means that it does not have a note_length child. We follow procedure as above, instantiating a Note or Rest object using the default value of one for durationMultiplier.

When we exit a multi_note, we will instantiate a Chord object that contains the Notes. According to the new specifications, the first note of a chords dictates the length of all other notes inside the chord. This is going to be handled as follows: as the first element to enter a Stack is the last to get out, the last Note to be popped out will be the first Note in the Chord, which is responsible for informing the whole Chord length. To do this, we will instantiate a temporary list and pop the Notes from the stack in order to prepend to this temporary list until we reach the SQ_BRACKET_OPEN token. At this point, we correct the each note.durationMultiplier by setting them each to the note.durationMultiplier of the first note in the temporary list. We then instantiate a Chord using this temporary list.

When we exit a note_element, we pop the object on the top of the stack in order to check if it is a Rest, Note, or Chord. If it is a Chord or Rest, then we push it back onto the stack. if it is a Note, we instantiate a single-note Chord object and push that onto the stack. We now only have Chords and Rests!

When we exit a tuplet_elt, we pop until we find a tuplet_begin token on the stack. Everything before that on the stack is a Chord in the tuplet. We call chord.multiplyDurationMultiplier passing in the argument as follows: In the case of a triplet, you should play Chord for 2/3 of the original duration. For a duplet, 3/2 of the original durations. And for a quadruplet, 3/4 of the original durations. We then push the Chords back into the stack in the reverse order that we popped them.

When we exit an element node, we pop the first thing off the stack in order to examine it. We instantiate a temporary stack.

- ✓ Case 1: In the case of note_element or tuplet_element, we popped a Chord or Rest, which we push back onto the stack.
- ✓ Case 2: We popped a BARLINE token, which means that all the Chords and Rests that are not in a measure already are in this new measure. We continue popping from the stack until we pop something that is not a Chord or Rest (in which case, we put it back on the stack). If it is a Chord or Rest, we prepend it onto a temporary list using ArrayList.add(0, Object). We then instantiate a Measure Object passing in our temporary list of SoundUnits (i.e. Chords and Rests). We push the Measure Object onto the stack.
- ✓ Case 3: We popped a SPACE token from the stack. It has no meaning so we ignore it.

When we exit an abc_line we know that we have one of the following cases:

- ✓ Case 1: A string representing the name of the voice in the stack
- ✓ Case 2: All the measures in the line in the stack
- ✓ Case 3: Just treated the lyrics

Lyrics

When we exit valid_letter, we push it to the stack as a char.

When we exit lyric_text, we pop all the children from the stack, and prepend them as a StringBuilder. We must first check if the previous String on the stack ends with a "~" or /-." If it does, we concatenate the two Strings and change the "~" to a " " and the "\-" becomes a "-", and push the new concatenated String onto the stack. Otherwise, we simply push the StringBuilder to the stack as a String.
When we exit lyrical_element, we check its content.

- ✓ Case1: if it's a TILDE, then we pop the last String from the stack, append a "~" to the String, and concatenate the next lyric_text with this String.
- ✓ Case2: if it's a SPACE, then we pop the last String from the stack. We check to if the String already ends in a space; if it doesn't then we append the space to the end of the String and push it back onto the stack.
- ✓ Case3: if it's a HYPHEN, then we pop the last String from the stack. If the popped String ends with a space or a hyphen, we should push that String back onto the stack and instantiate and push the new String "-" to the stack. If it doesn't end with a space of a hyphen, we append a hyphen to the popped String and push the new String onto the stack.
- ✓ Case4: if it's an UNDERSCORE, then we push the "_" String to the stack. (This will help align the lyrics with the chords.)
- ✓ Case5: if it's a STAR, then we push the "*" String to the stack. (This will help align the lyrics with the chords.)
- ✓ Case6: if it's a SLASH_HYPHEN, then we pop the last String from the stack, append a "\-" to the String, and concatenate the next lyric_text with this String.
- ✓ Case7: if it's a BAR_LINE, then we push the String "|" to the stack.

When we exit lyric, we pop from the stack until it is not a String. We prepend these popped Strings to a List<String>.

When we exit abc_line, there is a List<String> representing the song lyrics and Measures. At this point, we pop the stack to get the List<String>. We then pop the stack until it is no longer a Measure on the top of the stack. We now have to align the syllables to the Chords. In order to this, we call Measure.applyLyrics() on all the Measures we popped. Measure.applyLyrics takes a List<String> and matches it to the correct Chord. If it encounters a "|" then it makes the rest of the Chords in the measure have the "*" for its syllable. Measure.applyLyrics() returns the input List<String> without the Strings that were matched.

When we exit a barline, we know that anything that we have seen up to this point that is not already in a measure is in this new measure. At this point, we check if it is a malformed measure by adding up the durationMultiplier on each Chord and Rest that are on the stack, multiply that number by the defaultLength (from the abc_header) which should equal the beatsPerMeasure.

Handling errors in the input file

Our grammar is strictly accepting only well-formatted '.abc' files.

For instance: sometimes, 'bag' can mean a sequence of notes ('b', 'a' and 'g'), or it can just be a valid text input (as a name of a tune). To be able to handle these specific cases, we tried to create small meaningful tokens, and group them according to their behaviour. Inputs like 'm' or 'b' can have multiple meanings, and they were tokenized seperately. This way, we could create a completely functional, but still organized way to build our Abstract Syntax Tree using ANTLR. This is also ready for change, as we could improve the expressivity of our grammar by adding similar rules following the same strategy.

In order to create a fully functional MIDI file, together with its own lyrics, our lexer handles all the possible valid cases, so that if the user offers an '.abc' file with any kind of flaws (e.g.: inserting invalid characters as notes, or not offering the demanded header descriptions), it simply won't work. We build our

lexer and parser rules to be able to FAIL FAST! At the time the ANTLR faces an error, it throws a RuntimeException, which could later be handled to give the final user a more informative response.

One special invalid input that our program will accept is when a measure is malformed. If there are less than or greater than the number of beats required per measure, we will ignore bar lines when building the AST. As a result, malformed measures will be accepted.

Important notes

- When we are converting from the concrete syntax tree to the abstract syntax tree, we need to apply the key to the notes, otherwise we will have no way of differentiating (B) and (=B) after we've built our abstract syntax tree. This would be bad.

- Instead of creating a Concrete Class `Tuplet` that implements a Soundunit, we will apply length corrections as we convert from the concrete to the AST. For example, when we enter a triplet element (3ggg in the concrete syntax tree, we will multiply all the note lengths by ⅔.

- New specifications state that all notes in a chord last the same duration, therefore they have the same durationMultiplier. We will fix the durationMultiplier of the other notes in the chord at the moment we exit the Chord in the concrete syntax tree. (If we somehow can't do this, we will fix it later when we are calling getSeqOfPreMidiNotes.)

- We want to distinguish the cases between the notes F and =F. In order to do this, we are changing the Pitch class to have another argument, hasAccidental that is a boolean. If true, then it is has an accidental (including the natural accidental). If false, it does ot have an accidental. We also need to overwrite the .equals() method in order to check the hasAccidental boolean.

- We do not check for malformed measures. On Piazza (@494), Jared Wong stated that we must accept measures that contain less than the number of beats required. Since we must accept less than the number of beats, we have to decided to accept more than the number of beats as well.

- When we are aligning syllables to the Chords, WE MUST CHECK IF THE STRING WE SEEK TO ALIGN IS IN FACT THE String "|". If so, we should skip to the start of the next measure. Any unaligned Chords in the remainder of the current measure should have a "*" String.

2. Describe how you will take a representation of the input (e.g., your AST) and transform it into a format that you can cleanly play using SequencePlayer.

An ABCMusic instance is built using the concrete tree given by the Grammar. An ABCMusic instance contains the beatsPerMinute and ticksPerBeat necessary to instantiate a SequencePlayer. ABCMusic gives us a list of Notes, which has the necessary information (Pitch, startTick, and duration) to create each MIDI note in the SequencePlayer. Also, ABCMusic gives a list of Syllables, which has each syllable that will be shown on screen, and also the tick when it should be shown.

3. List the components of your system that you believe can and should be tested. For each of these components, describe your testing strategy and describe at least three specific test cases you plan to have.

```
**ABCMusic:
(immutable)
Attributes:  listOfVoices = List<Voice>;
             ABCHeader = ABCHeader Object;


Methods:  getSeqOfPreMidiNotes(): @returns List<PreMidiNote> to feed into the Sequence Player
                                  in Main.play()
          getSyllables(): @returns List<Syllables> to feed into the Sequence Player in
                          Main.play()
// NOTE: There might be a design change in which one method, getSeqOfPreMidiNotesAndSyllables
which returns a Map<PreMidiNote, String>

Private Helper Methods:
             // uses the beats per minute and what note gets the beat and the default length
             // in order to convert the soundunits to a PreMidiNote
             // with a Pitch, startTick, numTick
             // @param listOfAllRawSoundUnits the list of SoundUnits
             // @return List<PreMidiNote>
             applyMeterAndDefaultLength(List<NoteOrRest> listOfAllRawSoundUnits) {



**ABCHeader
(immutable)
Attributes:   beatsPerMinute = int;
              ticksPerBeat = int;
              beatsPerMeasure (**numerator in meter)= int;
              whatNoteGetsTheBeat (==1/denominator in meter)= double
                     // e.g. 4/4 meter -> (1/4), /2 -> (1/2) -> /1 -> (1/1), /8 -> (1/8);
              defaultNoteLength = double;
              key = String; // this is for the user's convenience ONLY; key has already been
                     applied
              composer = String;
              title = String;
              indexNumber = int;
Methods:      getTicksPerBeat(): @returns ticksPerBeat used to construct a Sequence Player in
                                 Main.play()
              getBeatsPerMinute(): @returns beatsPerMinute used to construct a Sequence
                                 Player in Main.play()
```

**Voice
(immutable)
Attributes: listOfSections: List<Section>
Method: getListOfSections(): @returns a copy of the listOfSections

**Section:
(immutable)
Attributes: listOfMeasures: List<Measure>;
Methods: getListOfMeasures(): @returns a copy of the listOfMeasures

**Measure:
(mutable)
Attributes: listOfSoundUnits: List<SoundUnit>;
endingBarLine = String; (can be “|”, “||”, “|]”, “[|”, “:|”, “|:”)
Methods: getListOfSoundUnits(): @returns a copy of the listOfSoundUnits
setEndingBarLine(String barLine);
applyLyrics(List<String>): @returns a list of Strings excluding the Strings that weren’t matched.

**SoundUnit (interface) = Chord + Rest
Attributes: getDurationMultiplier(): @returns an int that tells the numTicks a

**Chord: (implements SoundUnit)
(mutable)
Attributes: listOfNotesInChord = List<Note>;
syllable = String;
chordDurationMultiplier = double;
Methods: Constructor(List<Note>)
getSyllable(): @returns String
getListOfNotesInChord(): @returns a copy of listOfNotesInChord
getDurationMultiplier(): @returns chord DurationMultiplier
multiplyDurationMultiplier(double:factor): Mutator method. For each note in listOfNotesInChord, we call note.multiplyDurationMultiplier(factor). Also, chordDurationMultiplier = chordDurationMultipler*factor
setSyllable(String syllable): Mutator method. Sets this.syllable = syllable.

**Note:
(mutable)
Attributes: pitch = Pitch;
durationMultiplier = double (defaults to 1);
Methods: getPitch(): @returns a copy of the Pitch
getDurationMultipler(): @returns durationMultiplier
multiplyDurationMultiplier(double:factor): Mutator method. durationMultiplier = durationMultipler*factor
setDurationMultipler(double:newMultipler): Mutator method. durationMultipler = newMultiplier

Rest: (implements SoundUnit)
(immutable)
Attributes: durationMultiplier = double(defaults to 1);
Methods: getDurationMultipler(): @returns durationMultiplier

PreMidiNote:
(immutable)
Attributes: pitch = Pitch;
startTick = int;
numTicks = int;
Methods: getPitch(): @returns Pitch
getStartTick(): @returns startTick
getNumTicks(): @returns numTicks