Design

How Lexer and Parser work:

After receiving an abc-file, we use lexer to tokenize the file into tokens separately for the header and music part of the abc-file, we parse the header tokens at this stage, waiting for use in main. For the second part music, the parser parses the tokens into elements (which are pieces, each corresponding to one section of one voice). As shown below, each node has its own method getNumTicks(), by summing up all ticks played before the node with the same voice, the SequencePlayer can synchronize the playing of different voices.

Parser Design:

There are 1 or more several voices. The parser first parses all tokens corresponding to voice 1, and then deal with voice 2, voice 3 separately: when the parser is parsing tokens of voice i (Vi), if it meets Vj, where j is not i, then the parser skips the tokens following Vj until it meets Vi; if it meets Vi, it parses all tokens following Vi; the parser stops parsing for Vi if the parser meets "]" which represents the end of voice Vi.

Now we are dealing with all messages within one voice. The voice's message is broken up into elements.

(If there is the presence of repetition, 1 element is everything within the same section between the start, represented by "|:" and the end represented ":|" or between the previous end ":|" and the end ":|" in case of no start "|:")

Each element is a tree or a node (in the language of AST), as explained in Abstract Data Type Section later. Each element is labeled with its voice and index (the order among elements with the same voice).

The SequencePlayer simultaneously plays different voices by adding elements within the same voice in the order of their indexes (latter element has all note ticks begin at the tick of the last note of the consecutive former element) repectively. We use getNote() method from Element interface to get the notes and use addNote method to add notes from elements consecutively into SequencePlayer for the sequenceplayer to play the music.

Lexer Error Detection (throw exception)

Throw exception if after tokenizing each unit after "|:" there are no ":|" to end.

Throw exception if there are no token specifying "K", or "L", "M", or "X" in the header part.

Parser Error Detection (throw exception)

Throw exception if the contents in a measure is empty (don't have rests in the measure). Throw exception if there are tuplets other than Duplet, Triplet, Quadruplet

Abstract Data Types:

As shown in the dependency diagram on the second page, our abstract data types are abc-music, a class, which is a list of elements; element, which is an interface with Abstract Syntax Tree structure, and consists of two cases: singlestream, which is a list of measure; repetition is a tree whose root is a singlestream, and whose children are either repetition or singlestream. Measure is a list of singlets and tuplets between two bars. For details, see the grammar.

AST structure for element

For each element, either every note is played exactly once, that is, it has no repetition, and we call it singlestream (a node in the language AST), or it contains repetition (as a Tree). Element interface has basic operators as an AST, as shown below. The only difference is that tree can contain more than 2 children, so having function as numChildren().

Mutability:

All our datatypes are immutable for the security of the data. For operations, see the methods in element interface, and methods in note.

Grammar:

We modified the grammar slightly under abc-music by deleting the class abc-line, creating the class measure, and refining the definition of chord (aka singlet).

```
abc-music ::= element+
element ::= repetition | singlestream | mid-tune-field | comment
repetition ::= "|:"? root  child(1) (child())+
root ::= singlestream
child(n) ::= nth-repeat (repetition|singlestream) ":|"
single ::= measure+
measure ::= (singlet| tuplet)+ "|" | "|]"
tuplet ::= dup-spec singlet singlet |tri-spec singlet singlet singlet | quad-spec singlet singlet singlet singlet
singlet ::= [? Note+ ]?
note ::= note-or-rest [note-length]
note-or-rest ::= pitch | rest
pitch ::= [accidental] basenote [octave]
octave ::= ("'"+) | (","+)
note-length ::= [DIGIT+] ["/" [DIGIT+]]
note-length-strict ::= DIGIT+ "/" DIGIT+
accidental ::= "^" | "^^" | "_" | "__" | "="
basenote ::= [A-Ga-g]
rest ::= "z"
dup-spec ::= “(2“
tri-spec ::= ”(3”
quad-spec ::= “(4“
barline ::= "|" | "||" | "[|" | "|]" | ":|" | "|:"
nth-repeat ::= "[“DIGIT
```

Operations:

(Main

public method

constructor: Parser(Lexer lexer)

observer: Tree genAST()

)

Element
public method
constructor Tree(Root root, Tree[] children)
observer: Singlestream getSinglestream(), Tree getRoot(), Boolean hasChildren(), Int numChildren(), Tree getChildren(int index), Tree[] getAllChildren(), NoteInfo[] getNoteInfo(), override String toString(), Note getNote()
Visitor class (use visitor pattern)

Singlestream:
public method constructor singlestream(int note, int numTicks) obeserver getsinglestream(), getNumTicks(), Accept(VisitorElement<E> v)
Note:
public method constructor Note(pitch/rest, int Ticktime, int Tickduration)

Testing Strategy and Cases

We are going to have JUnit test to test all the modules from lexer and parser to specific datatype. Also, we are going to have integration test to test how different modules work together.

Junit Tests:
Note:
Case1. Test if the note has 1 note of duration a.
Case2. Test if the note has 1 note of duration b.
Case3. Test if the note has 0 note (it is a rest) of duration a.

Tuplet:
Case1. Duplet: 2 notes in the time of 3 notes
Test if the duration of a duplet equals to the duration of 3 notes.
Case2. Triplet: 3 notes in the time of 2 notes
Test if the duration of a Triplet equals to the duration of 2 notes.
Case3. Quadruplet: 4 notes in the time of 3 notes
Test if the duration of a Quadruplet equals to the duration of 3 notes.

Singlet (Chord):
Case 1. Test if there is only 1 note being played at one time (a “Chord” of 1 note)
Case 2. Test if the singlet is an actual chord with multiple notes (more than 1 note played at the same time)
Case 3. Test if the singlet is a rest (0 note being played at that time)

Measure
Case 1. Test if the duration of all the notes and the rest in internal(non-boundary) measures equal to meter which is the length of the measure
Case 2. Test if the measure has the open and close bar

Case 3. Test if the Node in the measure are legal input

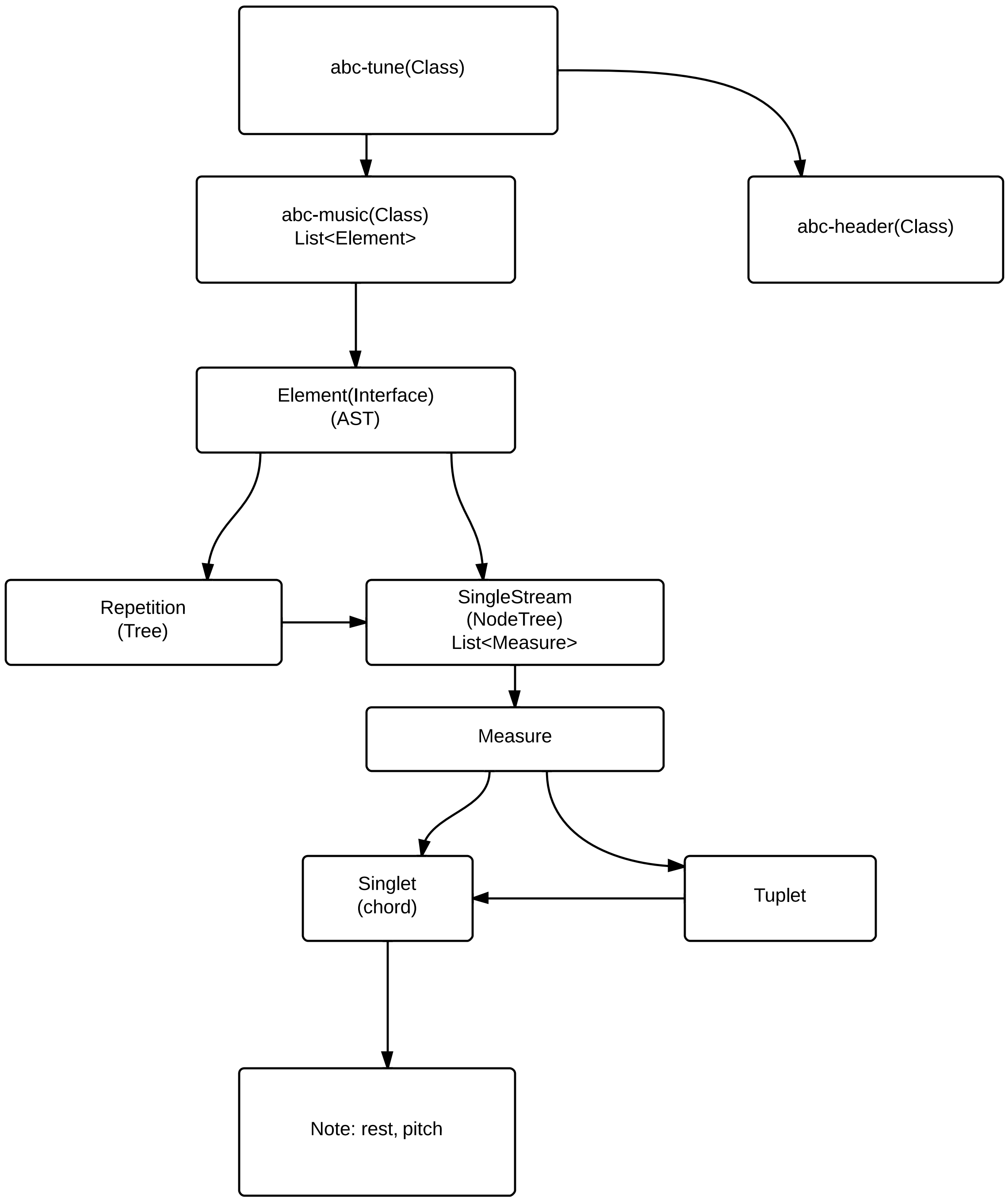
Repetition:
Case 1. Test if the music has a stream of singlets in which every singlet is played once (0 repetition)
Case 2. Test if the music has 1 repetition, create a tree with children as a stream of singlets
Case 3. Test if the music has n repetition (n>1), create a tree with children as a repetition tree(n-1)

Header
Case 1. Test if the header include all the necessary fields.
Case 2. Test if each field in the header occurs on a separate line.
Case 3. Test the order of 2 specific field:
    The first field in the header must be the index number('X')
    The second field in the header must be the title ('T')
    The last field in the header must be the key ('K')

Lexer:
Case 1. Test if all individual inputs corresponding to each token can be tokenized
Case 2. Test if the .abc with valid tokens can be tokenized to the different types of token we defined
Case 3. Throw exception if any part of the file cannot be tokenized

Parser:
Case 1. Test if the AST parsed by the parser is the same as what we expect
Case 2. Test if the order of the tokens obeys the grammar
Case 3. Throw exception if any part of the input cannot be parsed.

Integration Tests:
Case 1. Test if the .abc can be passed into lexer to be tokenized
Case 2. Test if the output of the lexer can be the input of parses
Case 3. Test if the output of the parser can be the input of Sequencer


abc-tune(Class)
abc-music(Class)
List<Element>
abc-header(Class)
Element(Interface)
(AST)
Repetition
(Tree)
SingleStream
(NodeTree)
List<Measure>
Measure
Singlet
(chord)
Tuplet
Note: rest, pitch