Revised Design

Design Changes

We rename the interface Element as AST to emphasize its role as the abstract syntax tree, and the two classes implementing the interface as ParentTree (whose children are the two variants) and NodeTree. We remove the data structure measure though we use it as an imaginary structure when dealing with temporary accidentals. We use a numerical structure for the tokens. We remove many methods in ParentTree and NodeTree, since in 6.005 subset of abc grammar, each ParentTree contains exactly two children. We remove some visitor-pattern-like operations because we no longer use the imaginary ADT they are operating on; besides, we reduce most of programs to be linear, which is easier to read, and easier to understand.

We provide more details tp explain how each part works.

How Lexer works

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. Our token is an object with nine fields: type, string of token, basenote, numerator, denominator, notelength, accidental, octave, chord, where the last fields are of integer type. We parse the header tokens at this stage, waiting for use in main. We won't explain in detail about how lexer processes information in the header since it is quite standard.

To tokenize the music body, first, we match the substrings to one of the types of the token according to the grammar below for the musicbody. Then for each token, the lexer computes the corresponding field values. For tokens other than pitches(notes) and rest, the last seven fields are set to be 0. The basenote are set to be 0, 1, 2, 3, 4, 5, 6, 7 for A, B, C, D, E, F, G, z respectively. E.g. for "a", the basenote is "A", and the octave is 1. (Our notion here is compatible with the convention in sound/Pitch.java). The chord number is nontrivial for a note or rest when it appears as the first rest-or-note in a chord, denoting the number of rest-or-notes inside the chord. Chord number is an important field for calculating the actual timing for the singlet following the chord in ASTtoPlayer. The numerator and denominator are those of the original notelength of the note or rest, e.g. for 1/ , numerator =1, denominator =2. Given the list of all denominators, it computes the tick per beat, so obtain the actual notelength of each note or rest, which is the number of ticks for the duration. We also adjust the notelength of notes in n-tuplets by multiplying 3/2 if n =2, (1-1/n) if n = 3,4. Finally, we create an arraylist of token for each voice, consisting of all tokens for the voice, and pass the arraylist of such arraylists to parser. There are 1 or more several voices. When the lexer is dealing with 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 appends all tokens following Vi; the lexer stops appending for Vi if it finds no more lines for Vi which represents the end of voice Vi.

Our convention for header and comments are as followed. First, we require X,T,K to be the first, second, last field of the header resp. The other fields can be in any order. The order of appearance of different voices can be arbitrary. We require no appearance of the same field other than the voice field V's. We also require the same voice appears only once in the header. We don't allow fields other than C, K, L, M, Q, T, X to appear in

header according to the grammar for header. Comments can also appear in the header. Comments should be in separate lines. Comments should start with symbol %.

How Parser works

The parser parses the arraylists for different voices separately. For the arraylist for one particular voice, the parser first checks if it has valid ending with barlines, and if it has, breaks it into different major sections, and build an AST for each major section. Breaking into major sections is favorable because the repetition cannot go across the major sections, so the parser can work modularly. In each major section, the parser checks if it contains any variants. If it does not, then apply ParseRepeat on the whole section; if it does, then it must contain exactly one root (the common part) and two variants according to our subset of abc grammar, apply ParseRepeat on each of them. The ParseRepeat takes in an arraylist of tokens, and output an arraylist of tokens equivalent to the old one in the sense of representing a piece of music, but without repeat symbols; it also removes tokens other than notes and rests, e.g. barlines. In the tree classes, we override a method to convert the tree into an arraylist of tokens equivalent to the tree. For a ParentTree, that is traversing all elements in root, then left child, then root, then right child.

We make a special note: we do not deal with directly nested repetition, but we allow a root and the variants (aka children) to contain non-overlapping (non-nested) repetitions. We also don't allow a child to be a ParentTree, it can only be an arraylist of tokens. We do not check the meter, because we allow incomplete measures, and we assume nothing about the convention of our meter. If we want to check the meter, we can immigrate the method dealing with chord in ASTtoPlayer, and count the number of ticks inside each measure. Nonetheless, we require a measure to be nonempty.

Our convention of accidental is as follows: the key in header gives a key signature on the whole music, the effect is in consistent with http://en.wikipedia.org/wiki/Key_signature. The (temporary) accidentals apply within the measure and octave in which they appear, unless canceled by another accidental sign.

How ASTtoPlayer works

We apply addNote step by step, using different layers of loops. Since each token records its own numTick (in the language of SequencePlayer), which is the field notelength. By inheriting the tempo and ticks per beats from Parser (and the latter from Lexer), we obtain parameters for SequencePlayer. We are a bit confused about the description about beatsPerMinute and TicksPerQuarterNote. We assume here quarter note is not actually quarter note, but a beat; and a beat is equal in duration to the default note length (indicated by the field 'L'). So in our Parser, we have a variable called tpb (=TicksPerBeat).

Note that before adding the note, to be consistent with notions in SequencePlayer, we convert the basenote into string. The SequencePlayer simultaneously plays different voices. Main is written in a standard way.

Lexer Error Detection (throw exceptions)

Throw exception if there are no token specifying "X", "T", or "K" in the header part as the first, second and last field, and if they are in the wrong order.

Throw exception if there are same fields other than voices in the header; if the same voice appears in the header more than once.

Throw exception if there are substrings which cannot be tokenized, that is, invalid token. (e.g Throw exception if there are tuplets other than Duplet, Triplet, Quadruplet; wrong accidental pattern, such as ^_C)

Parser Error Detection (throw exceptions)

Throw exception if a measure is empty.

Throw exception if there are directly nested repetition.

Throw exception if a major section contains only one variant.

Throw exception if some voice ends with non-barline (barline or endrepeat) .

Throw exception if the size of accidental is greater than 2.

Abstract Data Types

As shown in the dependency diagram on the second page, our abstract data types are abc-music, which is expressed as a sequence-of-voice-forest, which is an arraylist of voice-trees. A voice-tree is an arraylist of ASTs for a particular voice; AST, which is an interface with Abstract Syntax Tree structure, and consists of two cases: NodeTree, which is an arraylist of tokens; ParentTree is a tree whose root is an arraylist of tokens, and whose children are arraylists of tokens. (Each AST is built from a major section, and a major section is a list of measures; these in the brackets are not our ADTs). The fundamental ADT in our project is the token, which is an object with nine fields.
For details, see the grammar. ArrayList is mutable, hence to make it thread safe, we use Collections.unmodifiable. We also make some fields of token to be final. Token has method assertTokenEquals, We have operations on a major section, such as ParseRepeat (find the equivalent list of tokens without repeating symbols), ValidEnding (check if end with generalized barlines), and booleans to check if it has variants, and if validly. For first variant, we have Complete (complete the end with a barline).

AST structure for element

For each AST, either every note is played exactly once, that is, it has no repetition, and we call it NodeTree (where we already compensate repetitions without variants in ParserRepeat), or it contains repetition root as a ParentTree.

Grammar

We modified the grammar slightly under abc-music by deleting the class abc-line, creating the imaginary class (we don't actually write such class, but we use the idea when dealing with temporary accidentals in parser) measure, and creating an imaginary token type singlet (which is either a single note or a chord). The bold are the tokens.

abc-music ::= SequenceOfVoiceForest
SequenceOfVoiceForest ::= VoiceTrees+
VoiceTrees ::= AST+
AST ::= ParentTree| NodeTree
ParentTree ::= repeatbegin? root variant1 variant2
root ::= (repetition|singlestream)+

variant1 ::= 1st-repeat (repetition|singlestream)* (singlet|tuplet)+ repeatend
variant2 ::= 2nd-repeat(repetition|singlestream)+
repetition ::= singlestream
singlestream ::= measure+
measure ::= (singlet| tuplet)+ "|" | "|]"
tuplet ::= tuplets singlet+ //with the right number of singlets
//note that there is kind of abuse of notation here, need to be careful when reading
singlet ::= Note| Chord
Chord ::= ChordBegin Note+ ChordEnd
Note ::= Pitch | Rest
Pitch ::= Accidental? Basenote Octave? Notelength?
Rest ::= "z" Notelength?
Octave ::= ("'"+) | (","+)
Notelength ::= [d*/?d+]
Accidental ::= "^" | "^^" | "_" | "__" | "="
Basenote ::= [A-Ga-g]
Tuplets ::= "([234]"
ChordBegin::= "["
ChordEnd::= "] "
Barline ::= "|" | "||" | "[|" | "|]"
RepeatBegin ::=":|"
RepeatEnd ::= "|:"
Nrepeat ::= "["[12]
Comment ::= %.*
Whitespace ::= " "

Testing Strategy and Cases

We are going to have JUnit tests to test all the modules from lexer and parser to specific datatypes. Also, we are going to have integration tests to test how different modules work together. Here are some examples we planned to have.

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 (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

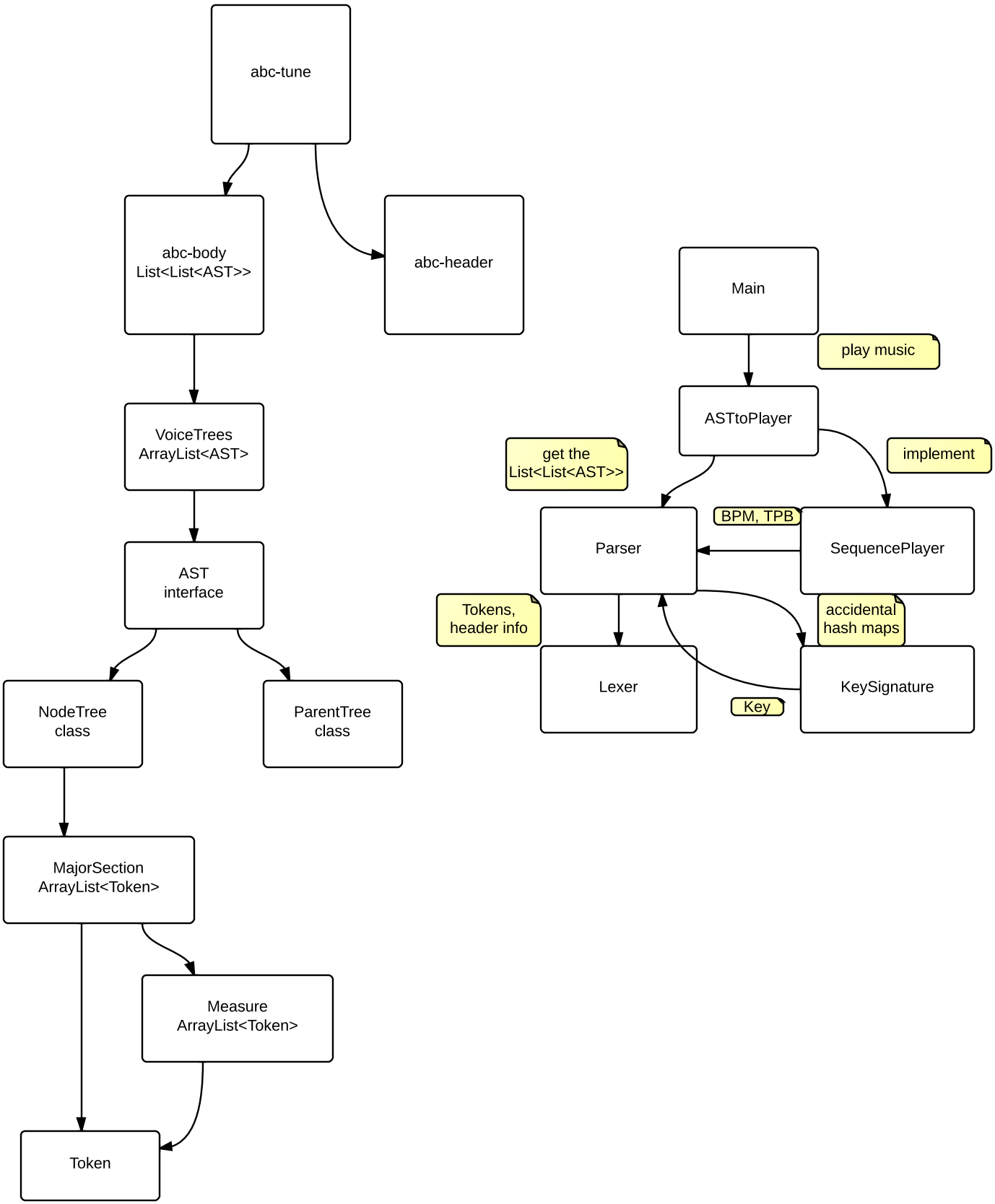
Repetition:
Case 1. Test if the music has a stream of singlets in which every singlet is played once (0 repetition)
Case 2. Test if the music has 1 repetition, create a tree with children as a stream of singlets
Case 3. Test if the music has n repetition (n>1), create a tree with children as a repetition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

Header
Case 1. Test if the header includes all the necessary fields.
Case 2. Test if each field in the header occurs on a separate line.
Case 3. Test the order of three specific fields:
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 parser
Case 3. Test if the output of the parser can be the input of SequencePlayer


abc-tune
abc-body
List<List<AST>>
abc-header
VoiceTrees
ArrayList<AST>
AST
interface
NodeTree
class
ParentTree
class
MajorSection
ArrayList<Token>
Measure
ArrayList<Token>
Token
Main
play music
ASTtoPlayer
get the
List<List<AST>>
implement
Parser
BPM, TPB
SequencePlayer
Tokens,
header info
accidental
hash maps
Lexer
Key
KeySignature