Design

Qian Lin, Chen Lian, Pavel Muravyev

The project can be break down into the following parts:
1. Read .abc file and break input strings into header and music.
2. Lex headerand, determine K, V, M, L, Q, and calculate from them beatsPerMeasure, beatsPerMitute and other relavant quantities.
3. Define Token for music according to our grammar. Lex music into token list.
4. Parse the token list generated by Lexer into an AST. Deal with accidentals within a measure. Note length given in units of L.
5. Evaluate AST to generate a list of (Pitch, start_tick, duration_tick) so that SequencePlay can play. Need to figure out ticksPerQuarterNote from the note lengths in AST, and deal with global accidentals specified by key signature.

The following design description will describe how we accomplish each part.

1. Read file and separate header/music

General description: player.Main.main promote user to input a .abc file name, and attemps to open the file. If successful, reads the file line by line and append to a List<String> inStream. After finishing reading and closing file, separate inStream into two subLists by field “K”. The first List<String> is fed to header lexer, and the second to music lexer.

special libraries: will import java.io.*.
input: file name (String).
output: two string lists, one containing lines from the file for header, the other for music.

Datatype: No new datatype needed.

Throw exceptions:
1. File open/read fails.
2. field “K” do not exist.
3. Either subList is empty.

Test:
1. File operations (open an non-existing file, open an existing file, read from an empty file, read from a normal .abc file.)
2. Separating into header/music (List<String> does not have K field, have K field at the beginning, have K field at the end.)

2. Header Lexing

General description: Extract information about key, default length of a note, meter and tempo from header. Check syntax validity for each line and field value. Check whether the first field in the header is index number ('X'), the second field in the header must be the title ('T') and the last field in the header is the key ('K'). Interpret K, V, L, M, Q values.

input: List<String> representing the header *output:* Header

Datatype:
1. Header: immutable

```
public class Header {
	private final int[] keySignature; //saccidentals derived from key signature for A-G respectively.
	private final Map<String, Integer> voice; //from voice label to index.
	private final int numVoices;  //total number of voices
	private final Rational L;  //unit length. positive
	private final Pair<Integer,Integer> M; //meter. positive
	private final int Q; //tempo, or number of L per minute. positive
	// initialize all fields of Header. If not specified, set to default value.
	public Header(List<String> input) {}
	private List<Pair<Character,String>> HeaderLexer(List<String> input) {}
	//Observators
	public int getQ() {}
	public Pair<Integer,Integer> getM() {}
	public Rational getL() {}
	// return index of voice v. If v is not in voice, throws RuntimeException.
	public int getVoiceIndex(String v) {}
	// return accidental value of note c. c must be within A-G.
	public int getAccidental(char c) {}
	public int getNumVoices() {}
}
```

Helper class:
1. Rational: immutable. based on code from http://introcs.cs.princeton.edu/java/32class/Rational.java.html (with minor modification)
Rep Invariant: gcd(num,den)==1, den!=0. throws RuntimeException if trying to make den=0;

```
public class Rational {
	public final int num; // the numerator. Non-negative int
	public final int den; // the denominator. Positive int.

	// initialize num and den. Rep invariant: gcd (num, den)=1.
	public Rational(int numerator, int denominator) {}
	@Override
	public Rational clone() {}
	public boolean equals(Rational other) {}// return string representation of (this)
	public String toString() {}// return (this * b)
	public Rational times(Rational b) {}// return (this + b)
	public Rational plus(Rational b) {}// return (1 / this)
	public Rational reciprocal() {} // return (this / b)
	public Rational divides(Rational b) {} // [Helper] return gcd(m, n)
	private static int gcd(int m, int n) {}
}
```

2. Pair: immutable. use code from ps2, but do not need comparable.

```
public class Pair<X , Y >{
        public final X first;
        public final Y second;
        // rep invariant: first, second != null
        // make a pair
        public Pair(X first, Y second) {}
        @Override
        public boolean equals(Object o) { }
}
```

ThrowExceptions:

1. X, T, K did not appear at the right place.
2. syntax incorrect (for example lack ":" between field and value).
3. value invalid. For example denominator=0.
4. duplicated voice field

Test:

1. Pair: X, Y of different type (String (empty, non-empty), primitive types, rationals (0/1, n/1, 1/n, m/n)).

2. Rational:
a. constructor: illegal (denominator=0), 0/m (m>=1), 1/m, m/1, m/n (gcd(m,n)>1)
b. plus/times/divide: a pairs of all combination of {0/m (m>=1), 1/m, m/1, m/n (gcd(m,n)>1)}, including cases of denominator of the pairs are the same/not the same.
c. clone: check for immutability (changing the cloned copy doesn't mutable the original copy).

3. Header
a. Detects syntax error (X, T, K not in correct position; Unrecognized field; Invalid field value; Unrecognized character or missing character (e.g. ":") in field definition)
b. Get rid of comments;
c. Intepret key signature correctly (test all A-G major/minor)
d. Generate voice to index map and nVoice correctly (duplicated voice name; V field not input in consecutive lines; one of the voice name is empty).
e. Q; L, M is correct

3. Music lexing:

General description: Generate token lists from the input strings representing music (including voice field).
The lexer reads the file line by line. For each line,
1. Differentiate whether the line is a mid-tune-field, a comment or an actual music line.
2. Write in the token list of the voice the mid-tune-field specifies or the default voice if nVoice=1;
3. Eliminate white spaces and throw runtime exception for invalid white spaces (for example within a tuplet or a chord, white space between accidental/octave/length and the basenote or "z", and other unrecognized input character).

The lexer constructor: for each symbol group it determines its type, generate token list and calls Token with type and the string list representing the token (string list is needed for intermediary. For example a note main need 6 strings, the first three represents accident, basenote, octave, and the last three represents length in rational number). Lexer is responsible to matching type and string.

Notice that for Token, we group accidental, octave, basenote and length into one Token, which shares the same datatype as the Note class in the AST. Similarly rest also combines “z” and length. This makes parsing easier, and also means that syntax checking for individual note and rest syntax is done by Lexer.

{ For reference, here is a list of terminal and intermediarys recognized by lexer:

```
        Terminals:

        mid-tune-field- ::= field-voice
        comment ::= "%" text linefeed
        basenote::= [a-gA-G]
        DIGIT::=[0-9]
        /
        accidental ::= "^" | "^^" | "_" | "__" | "="
        [
        ]
        |, ||, [|, |]
        |:, :|
        octave ::= ("'"+) | (","+)
        (

        Intermediary:
        note-length ::= [DIGIT+] ["/" [DIGIT+]]
        rest::= z [note-length]?
        note ::= [accidental] basenote [octave] [note-length]?)
}
```

input: List of strings to tokenize and Header. Each string is a line read from a abc file consecutively.
output: A list of different lists of tokens. Each voice has its own list.

Datastype:
1. Token: immutable

```
public class Token {
        public static enum Type {
        tuplet_spec, nth_repeat, multinote_start, multinote_end,
        simple_bar, major_section_start, major_section_end,
        repeat_start, repeat_end,
        rest, note;
        }
        public final Type type;
```

```
private final List<String> text;
private final int value; //used for type nth_repeat and tuplet_spec
private final Note note; //used for type note. See parser for class definition
private final Rest rest; //used for type rest. See parser for class definition
@SuppressWarnings("serial")
//map from accidental symbol to integer representation
private static final Map<String, Integer> accidentalToInt = Collections
.unmodifiableMap(new HashMap<String, Integer>() {{
put("^^", 2); put("^", 1); put("=", 0);put("_", -1);put("__", -2);
}});
/** construct a Token given type and the string s representing the Token.
* the specific type must match string s as specified in grammar.
* matching are (lexer is responsible to checking this syntax)
* simple_bar --- {"|"}
* repeat_start --- {"|:"}
* repeat_end --- {":|"}
* major_section_start --- {"[|"}
* major_section_stop --- {"||"} or {"|]"}
* multinote_start --- {"["}
* multinote_end --- {"]"}
* tuplet_spec --- {"(2"} or {"(3"} or {"(4"}
* nth_repeat --- {"[1"} or {"[2"}
* rest --- {"z"(,"[0-9]*"(,"/","[0-9]*"))} () means optional
* note --- {"^^|^|=|_|__","[a-gA-G]","[']*|[,]*"(,"[0-9]*"(,"/","[0-9]*"))}
*/
public Token(Type t, List<String> s) {}

/**
* [Helper] Convert List<String> s representing a rational number to a Rational.
* s.size()=0, 1 or 3. if size=0, return 1/1;
* if size=1, first element represents an integer.
* if size=3, first element represents numerator, or an empty string which be default *
* means 1; second element is "/"; third element is a denominator,
* or an empty string which by default means 2;
*/
private Rational toLength(List<String> s) {}
/**
* [Helper] s represents note. s.size()=3, 4, or 6. s[0] represents accidental (must
* be "^^", "^", "=", "_", "__", or empty string which means no accidental
* is specified). s[1] represents basenote, [a-gA-G]. s[2] represents
* octave, ("'"+) | (","+), or empty string which means not octave is
* specified. if size>3, the rest represents length.
*/
private Note toNote(List<String> s) { }
public int getValue() { }
// return a new copy of this.note
public Note getNote() { }
//@return a new copy of this.rest
```

```
        public Rest getRest() { }
}
```

2. Lexer: immutable.

```
public class Lexer {
        private final List<Token>[] tk;
        //generate array of token list from music strings. each list representing a voice.
        public Lexer(List<String>){}
        //number of voices
        public int getLength() {
        return tk.length;
        }
        //get a clone of ith token list. i must be in [0,getLength()-1]
        public List<Token> getTokens(int i) {
        return new ArrayList<Token>(tk[i]);
        }
}
```

Throw Exceptions:

Token:

1. length of input string list for note and rest does not match expected value.
2. strings for length is not a valid number
3. basenote is not a valid character [a-gA-G]
4. accidental syntax incorrect
5. octave syntax incorrent
6. nth-spec and tuplet digit out of bound (only support [1, [2 and (2, (3, (4).

Lexer:

1. additional space
2. invalid characters

Testing:

1. all kinds of tokens
2. eliminate valid white spaces
3. throw exeptions for invalid white spaces within tuplet and chord
4. throw exeptions for invalid input (unrecognized char)
5. Long sentence
6. Muliple voices

4. Parser

General Description: parsing is completed while constructing the AST hierarchically. The AST is a direct translation of the music input string, except that local accidentals (those within a measure) is assigned, e.g. | ^F F | will be stored as measure (note (F, +1), note (F, +1)) (this is just for illustration). Length for note and rest is expressed in units of L, so do not need to know any information from Header. The parser also chekcs for the rep invariant.

class(hierarchy): Music -> Voice -> MajorSection-> Section -> Measure -> [Note, Rest, Chord -> Note, Tuplet -> Note]* (all implements ABCmusic interface. A->B means A contains a list of B in one of A's fields.)

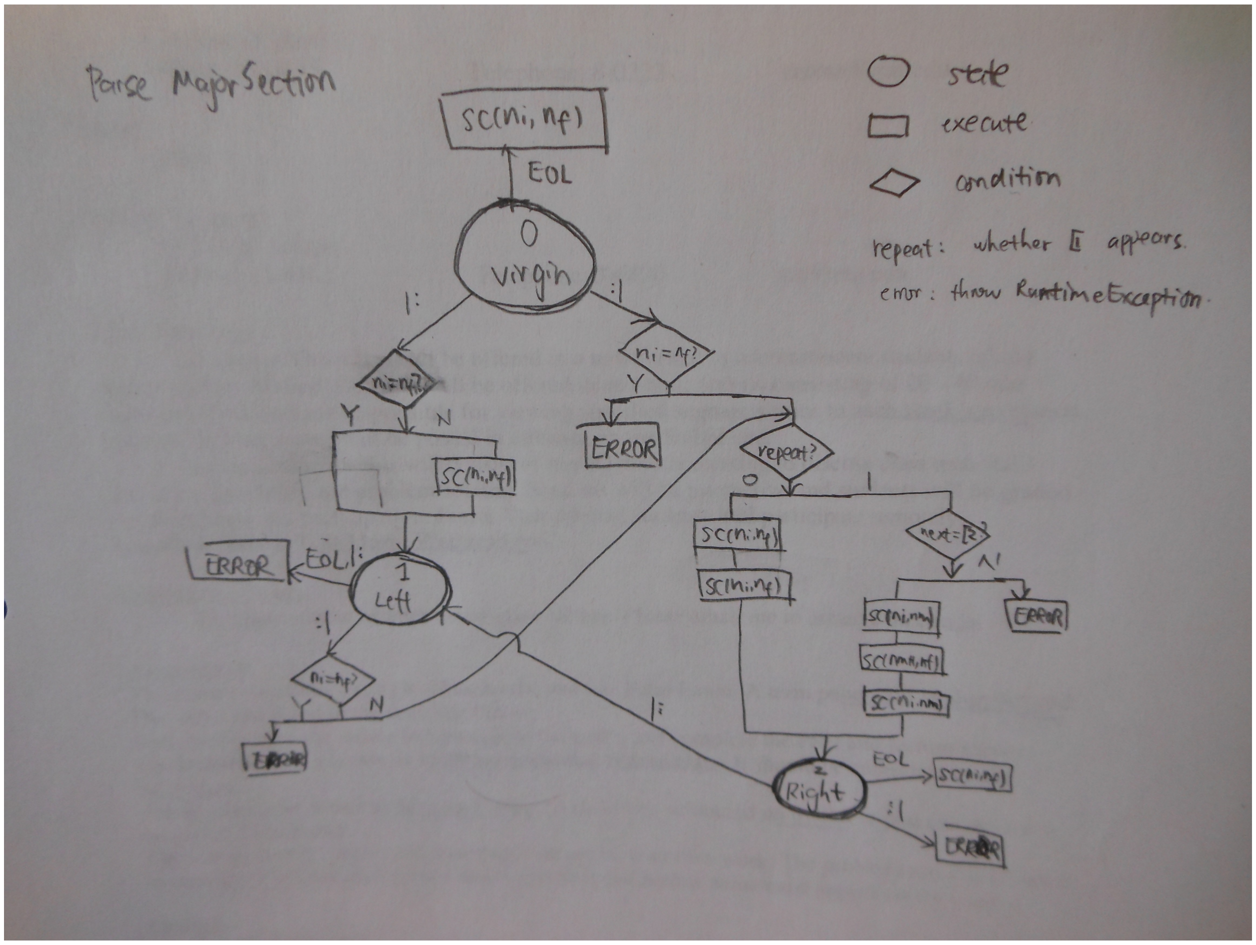
The following is a description of how the constructor for each class work, and how it detects syntax error in the token list:

<u>a. Music</u> constructor takes Lexer as input, and passes each token list in Lexer to Voice;

<u>b. Voice</u> constructor breaks the token list according to major-sections-bars ("[|", "||", "|]"), detects syntax error relating to major-section-bars, and pass each sublist (do not contain major-section-bars anymore) to MajorSection. Note that Voice reinforce that token list end with some sort of end-bar ("|]", "||", "|", :|"); it is acceptable to start with a major-section-start "[|".

<u>c. MajorSection</u> deals with repeat ("|:", ":|", "[1", "[2") using a state-machine (see Fig.1 below), and adds each section separated by "|:", ":|", "[1", or "[2" to its list of Section properly (e.g. |A| -> add(A); |: A :| -> add(A), add(A); |: A |[1 B :| [2 C | -> add(A), add(B), add(A), add(C);) Detects any syntax error related to repeat (nested repeats e.g. | A :| B :| or |: A |: B :| C :|, empty repeat section |::|, unmatched repeat (|: without :|, [2 before [1, or not [2 after [1). Pass each sublist (do not contain repeat anymore) to Section;

Fig.1 state diagram for parsing MajorSection. SC(ni, nf) measure add new Section(tk.subList(ni,nf)) to MajorSection.sections. There are 3 states (0 virgin, 1 left, 2 right) stored in field MajorSection.state. MajorSection.repeat denotes whether a "[1" was encounter before ":|".



<u>d. Section </u>breaks token lists into sublists according to bar (“|”). Detects syntax error related to “|” (e.g. start with a “|” because it means at some point “|” follows immediately after a repeat or major-section-bar). Pass sublist (do not contain “|” anymore) to Measure.

<u>e. Measure</u> adds Note, Rest, or group notes into Chord/Tuplet. Detect syntax error related to Chord and Tuplet (e.g. unmached “[“ and “]”, tuplet not followed by enough notes, notes in a tuplet has differnt length). Measure will also add accidentals to notes following the same note with specified accidental.

<u>f. Chord:</u> construct Chord given a list of notes. Detect syntax error if length of list is smaller than 2. Assign the length of the first note in the list to be the length of Chord. Prints an warming if two notes in a chord has the same pitch.

<u>g. Tuplet:</u> construct Tuplet given a array of notes, an int specifying the kinds of tuplet, and length of individual note. Detects syntax error if array length do not match tuplet-spec. Assign length according to tuplet-spec and individual note length.

<u>h. Note and Rest </u>is already constructed (and error-free) in Lexer, so just need to copy them from Lexer.

Parsing is done hierarchically. Since each layer works correctly, the overall AST is parsed correctly. Each layer eliminates possible syntax error related to cirtain token, so all syntax error will be detected during parsing.

input: Lexer
output: an AST (ABCmusic)

representation invariant:
voices have the same number of measures
corresponding measures in voices has the same length

Datatype:
1. Parser:

```
public class Parser{
        private final Lexer lex;
        private final ABCmusic music;
        public MusicParser(List<String> musicString, Header h){
                lex=new Lexer(musicString,h);
                music=new Music(lex);
                music.checkRep();
        }
}
```

2. ABCmusic interface: use Visitor patterns. Methods evaluate, checkRep

```
public interface ABCmusic {
        public interface Visitor<R> {}
        public <R> R accept(Visitor<R> v);
}
```

2.1 Music: mutable

```
public class Music implements ABCmusic {
        private List<ABCmusic> voices;
        public <R> R accept(Visitor<R> mu) {
        return mu.on(this);
        }
        //Generate a AST with music as the root.
        public Music(Lexer lex) {}
}
```

2.2 Voice: mutable

```
public class Voice implements ABCmusic {
        private List<ABCmusic> majorSections;
        public <R> R accept(Visitor<R> v) {
        return v.on(this);
        }
        //Break tokens into major section and add to majorSections using major-section-bar.
        public Voice(List<Token> tk) {}

}
```

2.3 MajorSection: mutable

```
public class MajorSection implements ABCmusic {
        private List<ABCmusic> sections;
        private int state = 0;  // index of state. see diagram
        private boolean repeat = false; // index of state. see diagram

        public <R> R accept(Visitor<R> m) {
        return m.on(this);
        }
        // tk is non-empty; end with ':|' or no bar
        //Break tokens according to repeats and add sections.
        public MajorSection(List<Token> tk) {}
}
```

2.4 Section: mutable

```
public class Section implements ABCmusic{
        private List<ABCmusic> measures;
        public <R> R accept(Visitor<R> s) {
        return s.on(this);
        }
        //Break tokens into measure according to bar.
        public Section(List<Token> tk) {}
}
```

2.5 Measure: mutable

```
public class Measure implements ABCmusic {
        private List<ABCmusic> elements;
        private Rational length=new Rational(0,1);
        private Map<Pair<Character,Integer>,Integer> accidentalList=new
        HashMap<Pair<Character,Integer>,Integer>();
        public <R> R accept(Visitor<R> m) {
        return m.on(this);
        }
        // group tokens in tk into one of note, rest, chord or tuplet and add to elemtns.
        Updates  // length by adding up length of all elements in this measure
        public Measure(List<Token> tk) {}
}
```

2.6 Chard: immutable

```
public class Chord implements ABCmusic {
        private final int value;
        private final Rational length;
        private final Note[] notes;
        public <R> R accept(Visitor<R> c) {
        return c.on(this);
        }
        // initialized spec, length, notes
        public Chord(List<Note> n) {}
        public Rational getLength() {}
}
```

2.7 Tuplet: immutable

```
public class Tuplet implements ABCmusic {
        private final Rational length;
        private final int value;
        private final Note[] notes;
        public <R> R accept(Visitor<R> t) {
        return t.on(this);
        }
        //initialize length. valeu, notes
        public Tuplet(int spec, Note[] n, Rational l) {}
        public Rational getLength() {}
        public int getValue() {}
}
```

2.8 Note: mutable

```
public class Note implements ABCmusic {
        private boolean hasAccidental;
        private int accidental;
        public final int octave;
        public final char value;
        private final Rational length;
        public <R> R accept(Visitor<R> n) {
        return n.on(this);
        }
        //initialize all fields
        public Note(char v, int o, int a, boolean b, Rational l) {}
        //return a new copy of Note.
        @Override
        public Note clone(){}
        //modify hasAccidental and accident. This is a mutator.
        public void setAccidental(int a, boolean b) {}
        public boolean getHasAccidental() {}
        public int getAccidental() {}
        public Rational getLength() {}
}
```

2.9 Rest: immutable

```
public class Rest implements ABCmusic {
        private final Rational length;
        public <R> R accept(Visitor<R> r) {
        return r.on(this);
        }
        public Rest(Rational l) {}
        @Override
        public Rest clone() {}
        public Rational getLength() {}
}
```

ThrowException: for any invalid syntax specified in General Description.

Test:

1. Each class implementing ABCmusic should be tested. Start with Rest and Note and move up the hierarchy. For example, Note input should test: base note (“a”, “A”), with octave and accidental, with length; Measure should test: input containing note, rest, chord, tuplet; one accidental; two accidental that cancels each other; multiple accidental on multiple basenote.
2. Parser should be tested with single voice, multiple voices, multiple voices that do not match.

5. AST -> SequencePlayer input

1. Determine the number of ticks per quarter note:
   - Visit every note
   - By note n, we will have calculated the least common multiple of the denomiators of the duration of notes 1..n-1; then the new lcm is the lcm(currentNote, lcm(notes[1..n-1])).
   - Once we have visited all the notes, the resulting value is the minimum ticks per note that allows our piece to be playable by SequencePlayer

2. Produce proper accidentals using Header.keySignature
   - Visit every section
   - In each section, call visitor.visit(note, previousAccidental) on each note consecutively; The visitor updates the accidental of the node it is visiting whose hasAccidental is false to match the key of the header file.

3. Set the starting tick to 0, and then visit every note (recursively -> visit Music, which visits MajorSection... etc) in order, and add it to Sequence player.
   - The length of the note, in ticks, is given by the number of ticks per quarter note times the length of the note times four (since it’s #ticks per *quarter* note).

For testing the conversion of the AST->SequencePlayer, we can mostly make use of the example .abc files provided.