1 Lexer and Tokens

Lexer takes a string representation of an ABC file (extracted through our main method) and creates tokens out of it. It first parses the header and then parses the body. It has 2 methods that return an ArrayList of Tokens for header and body respectively. The tokens are any of the following:

`ACCIDENTAL, BASENOTE, CHORDSTART, CHORDEND, NOTEMULTIPLIER, BARLINE, REST, REPEATSTART, REPEATSECTION, REPEATEND, TUPLET, VOICE, COMPOSER, TITLE, INDEX, KEY, METER, TEMPO, NOTELENGTH, ENDMAJORSECTION, OCTAVE`

We used a for loop to iterate over the string and matched the remainder of the string to a pattern corresponding to one of these tokens. This made it easy to go through the whole string without using too many conditional statements besides the regular expressions.

2 Parser

`Parser` is instantiated with a `Lexer` in it's constructor and within the constructor it generates a header object and a body object. By calling the `parseBody` and `parseHeader` methods a Player object is created from the body and header object and is returned by the `Parser`. The `parseBody` method first determines whether there were voices supplied in the abc file, if not it will create a default `Voice`. It then goes through the `Token` objects calling the appropriate methods to parse the `Token` objects and it assigns it to the current `Voice` which is the last seen `Voice`. Below is a diagram illustrating our `Parser` as a state machine.

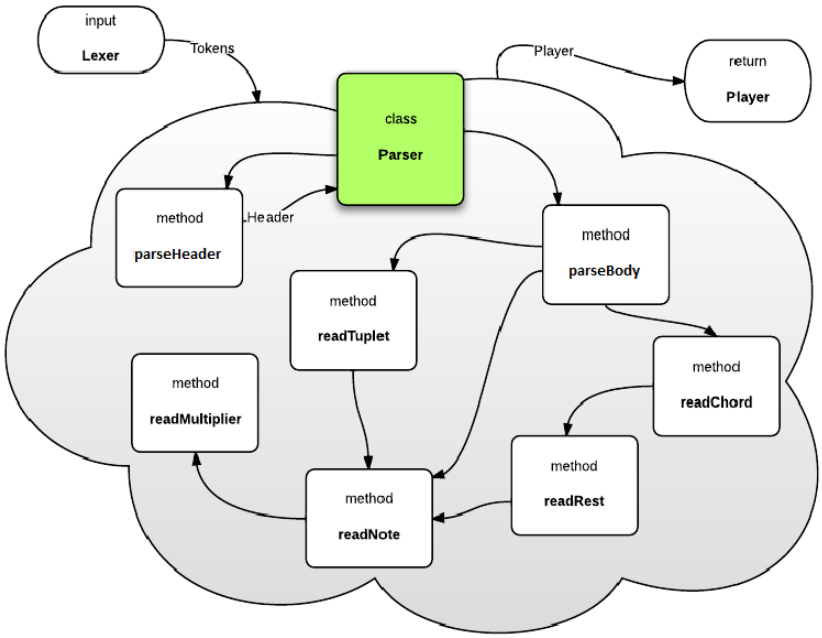

Figure 1: Graphical Representation of our Parser

3 Datatypes

`public enum KeySignature: KeySignature(int[] keyAccidentals, String stringRep)`

Representation of key signature types. The `keyAccidentals` array keep track of which `Notes` need to be transposed for a particular key signature, while `stringRep` contains a `String` representation of the name of the key signature in abc. it contains getter methods for `keyAccidental` objects, `stringRep`, and the `keyAccidental` type.

`public class Accidental: public Accidental(int intRep)`

Represents an accidental to be applied to a `Note`. `intRep` is the integer representation of the transposition of pitch by the `Accidental`.

`public class Fraction: pulic Fraction(int numerator, int denominator)`

This class represents a `Fraction`, to be used to store the meter and default note length. It has the following attributes:

- `int`numerator - Numerator of Fraction.
- `int`denominator - Positive denominator of Fraction.

`public abstract class MusicSequence:`

This class represents a valid musical construct of an abc file. It includes an accept() method for visitors that wish to perform operations on a particular `MusicSequence`. It includes a `startTick` attribute and getter and setter methods for this attribute.

```
public class Note extends MusicSequence:
public Note(char baseNote, int octaveMOdifier,
Accidental accidentalModifer, double NnoteMultiplier)
```

This class represents a not in an abc file using the following attributes:

- `char baseNote` - Represents the base note of the `Note`.
- `int octaveModifer` - Represents the particular type of `Accidental` applied to this `Note`
- `Accidental accidentalModifier` - Represents the particular type of `Accidental` applied to this `Note`.
- `Pitch notePitch` - `Pitch` object created with the above parameters
- `double noteMultiplier` - Represents a multiplier for the length of the `Note`, a multiplier of 1 indicates that the `Note` is of default length as specified in the header.

This class includes getter methods for the above fields as well as an `accept()` method for visitors.

`public class Rest extends Music Sequence: public Rest(noteMultiplier)`

This class represents a rest in an abc file. It has one attribute: the `noteMultiplier`, which acts similarly to the `noteMultiplier` in the `Note` class.

`public class Chord extends MusicSequence:`
`public Chord(List<Note>notes)`

This class represents a chord in an abc file. It hast the following attributes:

- `List<Note> notes` - Represent the `Notes` that make up the chord.

This class includes getter methods for the above fields, as well as an `accept()` method for visitors.

`public class Tuplet extends MusicSequence: public Tuplet(List<Note> notes)`

- `int tupletNumber` - Represents the type of tuplet $\in (2, 3, 4)$
- `List<Notes> notes` - Represents the `Notes` to apply the tuplet to

This class includes getter methods for the above fields, as well as an `accept()` method for visitors and a method to increment the current tick, as well as to get the current tick.

`public class Voice extends MusicSequence: public Voice(String voiceName)`

This class represents a voice in an abc file. It has the following attributes:

- `String voiceName` - Name of the `Voice`
- `List<MusicSequence> musicSequences` - `MusicSequence` objects that compose this `Voice`.

This class has getter and setter methods for these attributes, as well as methods to handle `Repeats`. It also contains methods to get and increment the current tick.

`public class Body extends MusicSequence: public Body()`

This class contains an `ArrayList` of `Voices` that make up the body of the abc piece. If no voice names are mentioned in the header, the list contains only one (default) `Voice`. It contains the following attributes:

- `List<Voice> voices` - List of voices present in the body of this abc piece. If no voice names are mentioned in the header, the list contains only one (default) `Voice`.

It contains methods to add the `ArrayList` of voices, to get the `ArrayList` of `Voice` objects, and an `accept()` method for visitors.

`public class Header: public Header(int indexNumber, String title, KeySignature keySignature)`

This class contains the header information that is included in every abc file. It has the following attributes:

- `String composer` - Name of the composer of the piece, default value "Unknown" represents the "C" field in the header.
- `int title` - Title of the piece, mandatory argument to the constructor as per abc specifications. Represents the "T" field in the header.
- `int temp` - Represents the number of default-length notes per minute. Default value is 100. Represents the "Q" field in the header.
- `Fraction defaultNoteLengthFraction` - Default duration of a note in this piece, default value set to $\frac{1}{8}$. The value is represented as a `Fraction` to preserve the numerator and denominator so the header can be faithfully printed by the `abcPlayer`. Represents the "L" field in the header.

- `KeySignature keySignature` - Key signature of the piece, mandatory argument to the constructor. Represents the "K" field in the header.
- `Fraction meter` - Represents the sum of all durations within a bar. Represents the "M" field in the header.
- `List<String> voices` - Represents the name of the voices in the piece. Represents the "V" fields in the header.

This class includes getter and setter methods for the above fields, as well as overridden `toString()` method.

`public class Player: public Player(Header header, Body body)`

This class contains a `SequencePLayer` that plays the `MusicSequence` objects contained within the `Body`. The `Header` and `Body` attributes are passed to the constructor while the `beatsPerMinute` and `ticksPerQuarterNote` attributes are calculated from information in the `Header`. It includes getter methods for these attributes, as well as methods to schedule and play the `Body`".

`public interface Visitor<R>:`

This class represents a generic visitor for a `MusicSequence`. It contains methods corresponding to each type of `MusicSequence`.

`public class Duration implements Visitor<Integer>: public Duration(Player player)`

This class uses the Visitor design pattern to calculate the duration of varius types of `MusicSequence` objects in the given player.

`public class MusicSequenceSchedule implements Visitor<Void>:`
`public MusicSequenceScheduler(Player player)`

This class uses the Visitor pattern to schedule `MusicSequence` objects in the supplied `Player`.

4 Design Decisions

1 We chose not to have a `DOUBLESHARP` or `DOUBLEFLAT token`; instead we had these operations on `NOTE` taken care of by the `parser`

2 We implemented the visitor pattern to handle both the duration and scheduling of our `MusicSequence`

3 We chose not to check for incomplete measures.

4 Even when there is no `Voice` specified in the header, we use a default `Voice`. This allows us to have one general implementation for all `abc` files.

5 We added a `Fraction` datatype to represent te meter and default note length.

5 Grammar

```
abcFile ::= abcHeader abcBody
abcHeader ::= fieldNum title comment* optionalFields* key
fieldNum ::= "X:"[0-9]+ end
title ::= "T":" text end
key:  ::= "K:" baseNote('#'|'b')?('m')?  end
```

```
optionalFields ::= composer | noteLength | meter | tempo |voice | comment
composer ::= "C:" text end
noteLength ::= "L:" fraction end
meter ::= "M":" ("C"|"C|"|fraction) end
tempo ::= "Q:"[0-9]+ end
voice ::= "V:" text end

abcBody ::= line+
line ::= (element* endOfLine) | voice | comment
nonBar ::= noteRep | tuplet| repeat| space
element ::= |?  nonBar |?
noteRep ::= note | chord
chord ::= "["note+"]"
note ::= noteType(noteMultiplier)?
noteType ::= pitch | rest
noteMultiplier ::= ([0-9]*)("/"[0-9]*)?
rest ::= "z" (noteMultiplier)?
pitch ::= (accidental)?baseNote(octave)?
accidental ::= "^ "|"^^ "|"="|"_"|"__"
baseNote ::= [a-gA-G]
octave ::= "'"+|"."+
tuplet ::= "("[2-4]noteRep+
repeat ::= repeatSimple| repeatAlternate
repeatSimple ::= (repeatStart noteRep+(barline noteRep+)* repeatEnd)
repeatAlternate ::= ( repeatStart noteRep+ (barline noteRep+)* [1
noteRep+ (barline noteRep+)* [2 noteRep+ repeatEnd)
barline ::= |
repeatStart ::= |:
repeatEnd ::= :|
majorEnd ::= || | |]
text ::= [.]
newLine ::= "\n"
space ::= " " | "\t"
comment ::= % text endOfLine
end ::= endOfLine | comment
endOfLine ::= newline | endOfFile
```

6 Revisions

1 We decided to make `MusicSequence` an Abstract Class rather than an interface so that we can could implement the `getTick()` and `setTick` methods.We

2 We decided that `Voice` should extend `MusicSequence` because it contains `MusicSequence`.

3 We added a `Body` class that contained all of our `MusicSequence` as opposed to having the player store an `ArrayList` of `MusicSequences`

4 We made `KeySignature` an `enum` type rather than a class because all `KeySignature` are constant and have specific attributes that we need to query efficiently; and an `enum` type is much better suited for this than a class. (This functions similarly to a hash map).

5 We modified the definition of `element` in the Grammar so that two bar lines without any `MusicSequence` in between would be prohibited.

6 We corrected the definition of `repeat` in the Grammar.

7 We separated the definition of a`barline` from `repeatEnd`,`repeatStart`, and `majorEnd`.

8 `Note` is no longer an interface and is instead a class that extends `MusicSequence`; this also means that `Rest` is a separate concrete implementation of `MusicSequence`. Additionally the `Pitch` class extending `Note` has been deprecated in favor of storing the pitch as an attribute in `Note` itself.

9 Removed the `Repeat` class and had the `Voice` class handle repeats because we needed access to the `MusicSequence` objects that came prior to the repeat in the `Voice` class in case we had to start the repeat from a different section.