Design Proposal

6.005 Project 1

Team: ahaidar, azizkag, ishabir

# Lexing

First, the lexer will go through the header of the file and create the following tokens:

- **C:** Name of the composer.
- **K:** Key.
- **L:** Default length or duration of a note.
- **M:** Meter.
- **Q:** Tempo.
- **T:** Title of the piece.
- **X:** Index number

Where each one of those tokens will contain the values attached to it. For example, if we get "C: Ahaidar" this will produce a token "C" with value "Ahaidar" and so on for the rest of the tokens.

As for the music part:

Tokenizer takes the string, and creates tokens. A first stage parser creates BAR objects that contain everything between bars. A second stage parser creates the voices and handles repetition. VOICE objects are then added to the sequence player via PLAYER, which adjusts based on the key signature, and everything is played. Essentially this is an i*nterpreter* way of doing it. Note that while adding things to the player we keep track of the tick that has been reached.

## TOKENS:

Tokenizer will produce the following tokens:

- basenote (C)
- accidental (_, ^)
- octave (,)

- length 2/3
- [
- ]
- tuplet_token (2 OR (3 OR (4
- bar (|)
- repeat_begin (|:)
- repeat_end (:|)
- |]
- voice (v.title)
- space

# Parser

Then a list of tokens is passed from the lexer to the parser which produces the AST containing the following data structures.

## DATA STRUCTURES:

Interface NOTESTRUCT
This will be implemented by NOTE, CHORD, and TUPLET

- int timeNumerator
- int timeDenominator
- int getDenom() - returns the timeDenominator (of any of its notes)
- int getNumTicks(ticksPerQuarter) : returns the number of ticks that this note (or, one of the tuplet notes) will occupy. This will only be called internally.
- int addToPlayer(startingTick, player): figures out how many ticks the note will occupy and adds it to the player. Returns the ‘tick’ that we are now at.

(Note: the time length will be in the form of timeNumerator / timeDenominator. For Tuplets, this will be for each individual note. )

NOTE implements NOTESTRUCT

- String basenote (a-gA-G | z (for rests) )
- String accidental
- String (or int) octave (Alternative: turn basenote and octave into an int)
- int timeNumerator
- int timeDenominator

(Note: the time length will be in the form of timeNumerator / timeDenominator )
(invariant:gcd( timeDenominator,timeNumerator ) = 1 to maintain simplest term)

<u>mutable:</u> for can change the length of the note later

CHORD implements NOTESTRUCT

- ArrayList<NOTE> notes

<u>mutable:</u> for containing an exposed mutable type

TUPLET implements NOTESTRUCT

- int Value (2, 3, 4)
- ArrayList<NOTE> notes (note: the timelength of the notes will not ignore the fact that it is in a tuplet (it will not be correct)

<u>mutable:</u> for containing an exposed mutable type

----------------------------------------------------------------------------------------------------------------

interface BARLINEOBJECT (a BAR or a bar-y token)

BARSIGNAL

- String value (or token value) (ex. |, |[, |[2, |:, etc. OR voiceChanger)

<u>immutable:</u> they are staying the same throughout.

----------------------------------------------------------------------------------------------------------------

interface BARSTRUCT (a BAR or VOICE)

- int getMinTicksPerQuarter() : returns smallest minTicksPerQuarter of all the notes under it.
- int addToPlayer(startingTick, player):
    - calls the PLAYER with everything it represents, in order. Returns the tick it reaches after adding itself.

<u>immutable:</u> they are staying the same throughout.

BAR implements BARSTRUCT, BARLINEOBJECT

- ArrayList<NOTESTRUCT> *notestructs*- this will contain the NOTESTRUCTS within that bar
  <u>immutable:</u> they are staying the same throughout.

VOICE impements BARSTRUCT (mutable)

- ArrayList<BARSTRUCT>
- append(BARSTRUCT)
- String name
- int startPoint : represents an index (in the ArrayList) - everything after this is in the most recent ‘group’.
- hitStart(): resets startPoint to the last index (= ArrayList.length)
- repeat(): adds to the ArrayList all the bars since the hitStart
- replaceLast(BAR): replaces the last bar with the received BAR argument

  <u>mutable:</u> we keep adding BARSTRUCTS after we create it.

## stage1Parser(ArrayList<TOKEN>);

Create an ArrayList<BARLINEOBJECT> - either BARs or BARSIGNALs everything into BARs and BARLINEOBJECT

Parser is an iterator, and will keep track of the index *i* in the list of tokens. Parser will fill in an ArrayList<BARLINEOBJECT>.

Parser will ignore all voice-related tokens. Parser will turn all bar-like tokens into BARSIGNALs. After every BARSIGNAL, it will call barConstructer, which will return a BAR (representing all objects until the next barline, and is made up of NOTE, CHORD, and TUPET objects, created by calling noteConstructer, chordConstructer, and tupletConstructer, respectively).

**BAR barConstructer(i):**

- takes index i of the Array that the parser is going through, moves through until the end of the bar, and constructs and returns the bar. i is updated in the process.
- create a new ArrayList<NOTESTRUCT>  and add  NOTESTRUCTS to it until the next

BARSIGNAL is observed. It will also create:
    - an ArrayList<String> naturals: list of basenotes with naturals.
    - an ArrayList<String> sharps: list of basenotes with sharps.
    - an ArrayList<String> flats: list of basenotes with flats.
- If the parser encounters a tuplet_start token:
    - it will call tupletConstructer(i),
- If the parser encounters a [ :
    - it will call chordConstructer(i),
- Else (a normal note is reached)
    - it will call noteConstructer(i)
- The returns of these methods will be added to the ArrayList<NOTESTRUCT>.
- When a new BARSIGNAL Is encountered, the parser creates a BAR with the NOTESTRUCTS it has gathered in the ArrayList, and adds this BAR to the ArrayList of BARLINEOBJECTS.

**TUPLET tupletConstructer(i);**

- Takes the current index and moves through tokenArray until a space is encountered. After saving the tuplet number (2,3,or 4) it calls noteConstructer until a space is reached, and will save the returned NOTEs in an ArrayList<NOTE>, which it will then use (along with the original tuplet number) to construct the TUPLET. The TUPLET is returned. i is updated in the process.

**CHORD chordConstructer(i);**

- takes the current index and moves through tokenArray until a ] is reached. It will call noteConstructer until a space is found, adding the returned NOTEs to an ArrayList<NOTE>. It will then construct a CHORD with this list of NOTEs and will return the CHORD. i is updated in the process.

**NOTE noteConstructer(i);**

- takes the current index and moves through tokenArray until a space is reached. It will make a NOTE object with the tokens (accidentals, base notes, octaves) it went through, and returns the NOTE. i is updated in the process.
- If the note does not have an accidental, its basenote is looked for in the three accidental lists (naturals, sharps, flats). If it is found in any, the appropriate accidental is manually added to it.
- If the note has an accidental, its base_note are added to the appropriate accidentals list (and removed from other accidentals lists, if it is there).

Once the ArrayList of BARLINEOBJECTS is created, it is passed on to the Stage2Parser.

## Stage2Parser(ArrayList<BARLINEOBJECTS>);

int repeatStartPoint = 0;

- This will constantly hold the index in the array of the last object that is either a |] or a |:

ArrayList<VOICE> voices;
ArrayList<String> voiceNames;

- voices and voiceNames must be created at the very beginning, and must either be available as global variables, or must be retrievable via some method.

REPEAT repeatConstructer(i);

The Job of Stage2Parser is twofold:

- First, it constructs and builds up the voices.
- Second, it fills up the voices with BAR objects, which it creates.

This parser is a state machine, with the state being the number(index) of the voice it is currently filling up.

Parser iterates through the list of BARLINEOBJECTS. If it encounters a voice object, it switches the currentVoice. If it encounters a |, it just adds the previous bar to voices[currentVoice] (which is a VOICE object). If it encounters a |: or |], it calls VOICE.hitStart(). If it encounters a :|, it calls VOICE.repeat() after adding the bar. (and calls hitStart())

If it counters a bar with [1, it adds the bar, calls repeat, then calls replaceLast(next Bar), calls hitStart and continues.

Parser returns the list of VOICEs.

## PLAYER

This class is what receives all play commands from the notes, mutates them as dictated by the key signature (adds sharps/flats if necessary) and then adds them to the actual SequencePlayer (a global object).

## OVERALL: PLAYING

We end up with everything contained in VOICE objects.

First, our main class calls the VOICE's getMinTicksPerQuarter(), which returns the minimum number of ticks per quarter necessary to sustain all notes within that VOICE. We call this for all VOICES in the array, and use the largest one as the *ticksPerQuarter* value. Then, we call addToPlayer(ticksPerQuarter) on all the VOICES, which sends, in order, all the notes to our PLAYER. This builds up our SequencePlayer. We tell it to play, and we're done.