aybose, katielee, meenasub

6.005 DESIGN MILESTONE

Design

- <class> KeyMap() (mutable)
  - the class that basically generates a map corresponding to our key signature
  - has a HashMap (originalKey) (immutable) that never changes
    - flat = -1, sharp = 1, natural = 0
    - example: key of G would have the map {G: 0, A:0, B:0, C: 0, D:0, E: 0, F:1,
    - not case sensitive (because it should be able to handle different octaves)
  - has a HashMap (accidentalKey) (mutable) that stores accidental nature for each note with a corresponding accidental in a bar
  - methods
    - initialize KeyMap given a String representing the key
      - using this convention
    - KeyMap.reset()
      - called after every bar
      - clears the accidentalKey HashMap
    - KeyMap.accidental(String note, int i)
      - should update the accidentalKey HashMap to have flat/sharp/natural for a given note (example barKey.set(note, i))
    - KeyMap.get(note)
      - checks accidentalKey map first. if not located in there, checks originalKey map.
      - for a given note, should return the integer value (-1, 0 or 1)
- <class> ParseHeader
  - reads in the .abc file and parses all relevant information from the header
  - methods:
    - getKeyMap() → instantiates and returns a Map of the key signatures given the key specified in the header
    - getTicksPerNote() → returns an int for the number of ticks per default note length
    - getTempo() → get tempo information
    - getVoices() → returns a list of Strings representing each voice
      - We only check for voices in the header
      - If no voice is listed, returns “SingleVoice”
- <class Token> (immutable)
  - initializes different tokens based on their Types
    - BASENOTE-- any basic letter A-G (not case-sensitive)
    - REST-- z

        - TIME--
        - ACCIDENTAL-- sharps, flats, natural (^, ^^, _, __, =)
        - OCTAVE-- octave shifts (string of commas and apostrophes)
        - MEASURE_BAR-- |
        - START_REPEAT_BAR-- |:
        - END_REPEAT_BAR-- :|
        - DOUBLE_BAR-- ||
        - END_BAR-- |]
        - START_BAR-- [|
        - NTHREPEAT1-- [1
        - NTHREPEAT2-- [2
        - TUPLET_SPEC-- (2, (3, or (4
        - OPEN_CHORD-- [
        - CLOSE_CHORD--]
        - VOICE-- voice token

- <class> Lexer
    - reads in the .abc file and creates a list of Tokens
    - take advantage of Java regex
    - methods
        - getTokens() → returns a List of Tokens
        - toString() → string representation of all of the tokens together
            - useful for debugging maybe

- <class> Voice (mutable)
    - handles the Elements associated with each part by adding them to a List
    - only check for voices listed in the ParseHeader
    - attributes:
        - int open: when we have an open repeat symbol
        - int ending: case of alternate endings?
        - int globalTick: increments the global tick as each element with a certain numTicks is added
    - .toString() → returns “Voice1” or “Voice2”, or whatever the voice is called in the .abc file
    - .addElement(Element e) → takes in an Element and adds it to the List of Elements, updates globalTick
    - .getElements() → returns the list of Elements for the voice part
    - .repeat(int closed) → copies the list from index int open to int closed, and appends it to the List of Elements
    - .alternateEnding()
        - repeats from start to nth-repeat
    - .eval(int globalTicks) / visitor pattern
        - should iterate through the List and evaluate each of the Elements in the

List

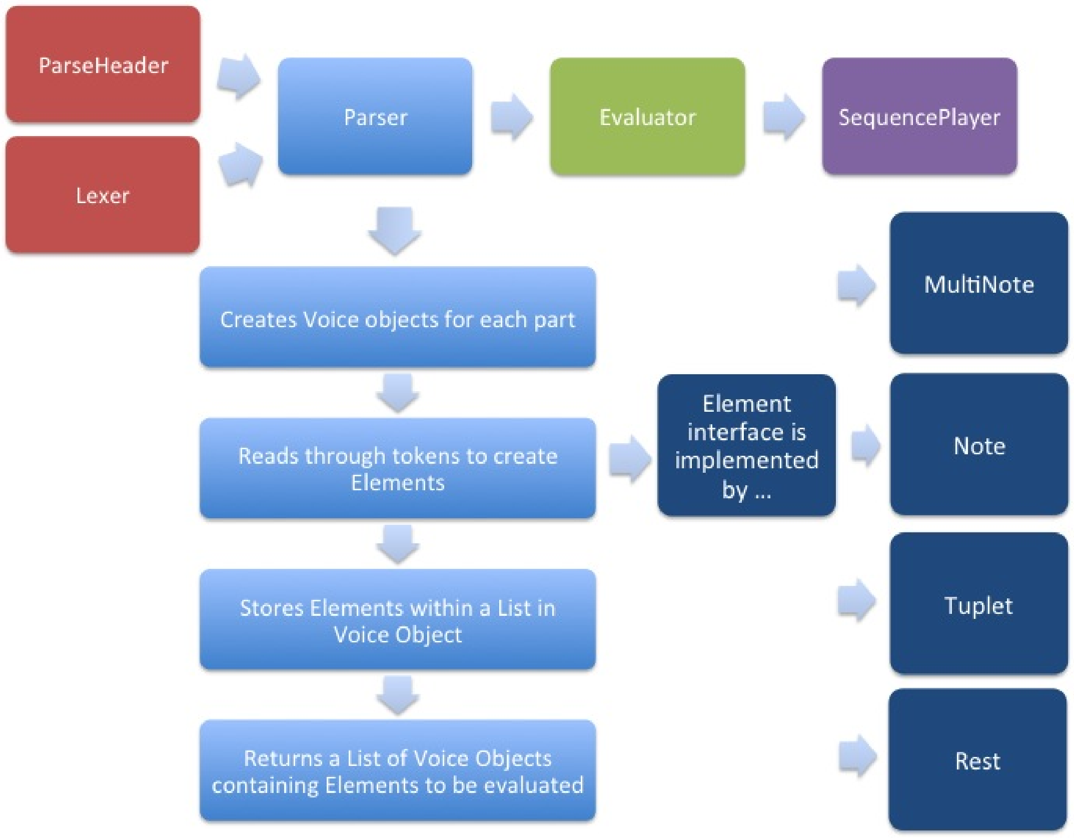
- <interface> Element (immutable)
    - implemented by the classes Note, Rest, TupletElement, MultiNote
    - common methods:
        - getLength() - returns length of each Note/Notes
        - .eval() / visitor pattern
            - eval should return a MIDI note or a set of MIDI notes that the player can play
    - <class> Note:
        - attributes: length, pitch, octave, accidental (these are set when Note is instantiated)
    - <class> Rest:
        - attributes: length
    - <class> TupletElement
        - attributes: length of tuplet in total, length of each note in the tuplet
        - instantiates however many Elements (Note, Rest, or MultiNote) are in the tuplet
            - ex: TupleElement(triplet) would be instantiate 3 Elements
    - <class> MultiNote
        - attributes: length of the note
        - instantiates Notes of the same length (however many notes are in the chord)

- <class> Parser
    - takes in Lexer and ParseHeader as input
    - reads through tokens, instantiating Elements with a calculated note elngth and adding them to a Voice object's List.
    - if we reach a bar Token, MapKey.reset() is called.
    - if we reach an nth repeat token, Voice.repeat() /alternateEnding() is called
    - if we reach a voice token ,we switch the current Voice that we are adding Tokens to
    - returns a List of Voice Objects, which are passed on to the evaluator
        - i.e. in a Parser.getVoices() method

- <class> Evaluator
    - Evaluates different voice parts one at a time, which will instantiate MIDI notes for each Element in each Voice.
    - each Element will have a .eval(int globalTicks, SequencePlayer player) method that adds a new Pitch.toMidiNote() to the player (or multiple notes in the case of a tuplet or chord).
    - Basically, we use interpreter pattern to keep track of globalTicks and update the player.

- ○ in the end, after all of the notes have been added, we can call player.play()

Process Flow Diagram

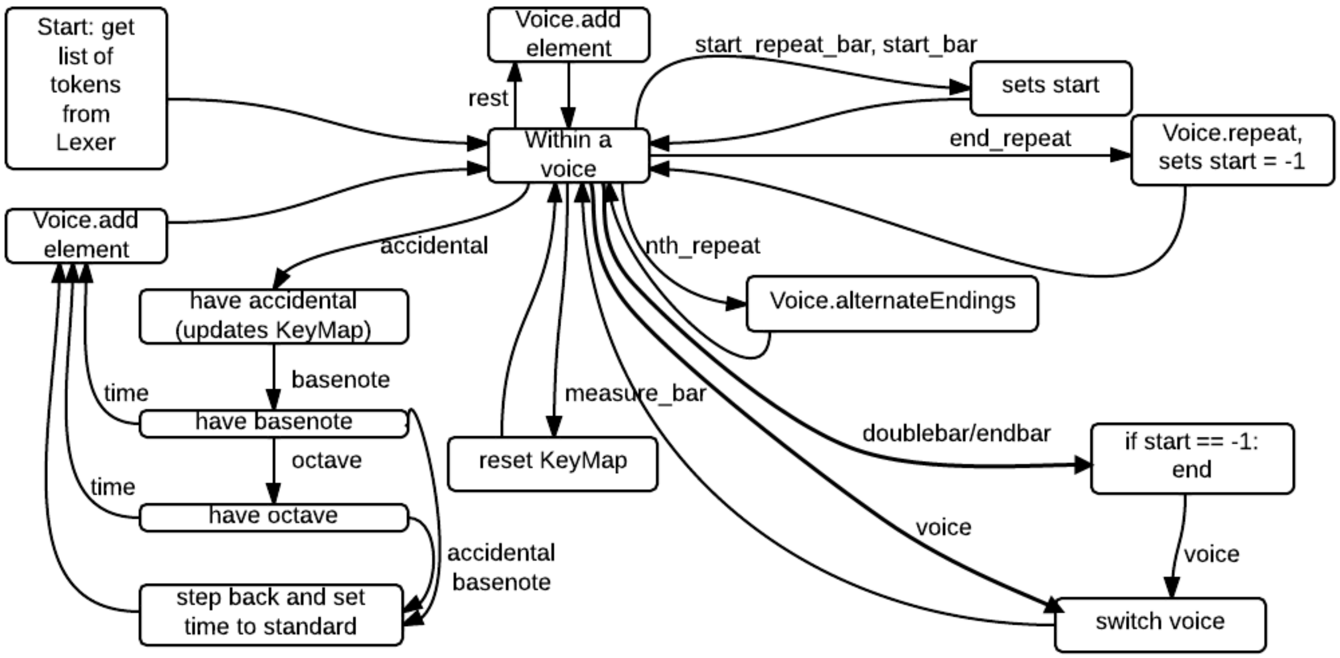

ParseHeader
Lexer
Parser
Evaluator
SequencePlayer
Creates Voice objects for each part
Reads through tokens to create Elements
Stores Elements within a List in Voice Object
Returns a List of Voice Objects containing Elements to be evaluated
Element interface is implemented by ...
MultiNote
Note
Tuplet
Rest

Modified Grammar

We changed :
element ::= note-element | tuplet-element | barline | nth-repeat | space
note-element ::= (note | multi-note)
to:
element ::= note | multi-note | tuplet-element | barline | nth-repeat | space

Parser State-Machine Diagram

With tuples and chords, the left side of the diagram will loop for x number of times for a tuple or chord. We omitted this from the diagram to keep it simple. Any unexpected tokens (in other words, no arrow from that state exists with that token) will throw an error.



Testing Strategy

We will test the following classes:

- Lexer:
  - Validity of each token class
  - Incorrect token handling
  - Whitespace handling
  - Linefeeds handling
  - Comment handling
- Parser:
  - Valid one-voice piece
  - Invalid one-voice piece
  - Valid two-voice piece
  - Invalid two-voice piece
  - Valid multiple-voice piece
  - Invalid multiple-voice piece
  - Chords
  - Tuplets
  - Repeat with
    - No special barVoice:
    - Start bar
    - Double bar
    - Start repeat bar

        - Alternate Endings
    - Octaves
    - Accidentals
    - Accidentals with different octaves
    - Different tempos
- HeaderReader:
    - getNumTicks
        - With different tempo listings
        - With tuplets
        - With chords
    - keySignature
        - Valid
        - Invalid
    - Voices
        - Single
        - Multiple
        - Not listed
        - Listed
    - No special bar
    - Start bar
    - Double bar
    - Start repeat bar
    - Alternate Endings
- Evaluator:
    - Same tests as parser for integration testing
- KeyMap:
    - All valid key signatures
    - Invalid key signature handling
    - Adding to accidentalKey
    - Getting from accidentalKey
    - Getting from originalKey