6.005 Project 1 Design Proposal
Team Member: Keren Gu, Michael Sanders, Shu Zheng

Table of Contents

I. Description

Given the problem that we ought to solve, the *abcplayer*, the following is a design proposal including the precise definition of the music-grammar and a general datatype organization.

The Input:

The input of the *abcplayer* is the file path, within the project abcPlayer/, to the music file. The input file format that we are responsible for are defined in the grammar below.

As an overview, the input file must contain a Header Section, where the first header must start with “X:” and second start with “T:”. Finally, the header section ends with “K:”. We will process everything after “K:” as part of the Body. Therefore if headers appear after “K:”, we will produce a wrong output sound sequence or throw exceptions.

In the Body section, we are able to handle regular note pitch, accidentals, octave-changes, chords, duplets, triplets, quadruplets, up to 2 repeats. We do not check if each measure contains the correct number of beats because we believe that’s the responsibility of the composer. We also do not handle pitch’’,,, , tokens like this will be interpreted incorrectly and thus providing a wrong output.

The Output:

The output is the voice output that plays the music file.

Implementation Flow:

The overall plan of attack is the following:

Input filename → SONGBUILDER → Input FileString
Input FileString → LEXER → List of Tokens
List of Tokens → PARSER → Piece of Music
Piece of Music → PIECE.GET_USEFUL_NOTES → List of Playable Notes
List of Playable Notes → MAIN → Output music.

II. Grammar

Grammar = Header Body

Header = X-line end-of-line T-Line end-of-line (Optional-Line end-of-line)* K-Line end-of-line

X-Line ::= X : [DIGIT+]
T-Line ::= T: [a-z]*
K-Line ::= K: [A-G] (#|b)? m?
Optional-Line ::= C-Line | L-Line | M-Line | Q-Line | V-Line*
C-Line ::= C: String
L-Line ::= L : ([DIGIT+] / [DIGIT+]) | [DIGIT+]
M-Line ::= M : ([DIGIT+] / [DIGIT+]) | [C] | [C\\|]
Q-Line ::= Q : [DIGIT+]
V-Line ::= V: String

String = any string
DIGIT ::= [0-9]

Body ::= (V-Line end-of-line **music** end-of-line)* | (**music** end-of-line) *
music ::= note-element | barline | nth-repeat | space

note-element ::= (note | tuple-element | chord)
note ::= note-or-rest (note-length)?
note-or-rest ::= pitch | rest
pitch ::= [accidental]? basenote [octave]*
accidental ::= "^" | "^^" | "_" | "__" | "="
basenote ::= "C" | "D" | "E" | "F" | "G" | "A" | "B"
| "c" | "d" | "e" | "f" | "g" | "a" | "b"
octave ::= ‘+ | ,+
rest ::= “z”
note-length ::= [DIGIT+]? ["/" [DIGIT+]?]?
tuplet-element ::= tuplet-spec note-element+
tuplet-spec ::= \\([2|3|4]
chord ::= \\[note+ \\]

barline ::= "|" | "||" | "[|" | "|]" | ":|" | "|:"
nth-repeat ::= "[1" | "[2"

end-of-line ::= comment | “\n”
comment ::= "%" String

III. Datatypes

class Piece:
- Header header
- Body body
- String fileName
- int beatsPerMinute
- int ticksPerQuarterNote
- List<NoteElement> getUsefulNotes: returns a list of playableNotes.
- Getters and Setters for the Piece Field.

class Header:
- int trackNumber_X
- String name_T
- String key_K
- String composer_C
- String length_L
- String meter_M
- int tempo_Q
- List<String> voices_V
- int numVoices
- Getters and Setters for every field.

class Body:
- List<NoteElement> notes
- Getters for notes.

Interface NoteElement:
- enum Type for the different types of NoteElement
- getType(): noteType
- toString(): String
- getDuration(): String
- getDecumalDuration(): double
- class Nplet implements NoteElement
  - enum npletType
  - npletType type
  - int numNotes;
  - List<NoteElement> elements
- class Chord implements NoteElement
  - List<NoteElement> notes
- class Rest implements NoteElement
  - String duration
- class Note implements NoteElement
  - char pitch
  - String duration (original beat-length, not the tick number)
  - enum type accidental
  - int octave
  - AccidentalType accidental

- int semitoneUp

//What is a SongBuilder: handles the reading of the file, and passing of data from lexer to parser to internal state representing the notes in NoteElement form. A SongBuilder object is multiple-use: after being constructed, should be able to process multiple input files sequentially
class SongBuilder
List<NoteElement> getnotes: outputs the notes in piece
constructor SongBuilder(): nothing happens // Use Default Constructor
Piece buildSong(String abcFileName): reads in abcFileName
class Main
Main.play plays the input file to the best of it's ability.
class Lexer
(normal lexer class and methods)
/** Constructor:
Lexer(String string)
/** lexate: uses string attribute and returns a list of tokens
List<Token> lexate(String abc): given valid abc file, returns a list of Tokens
class Parser
// Lexer lexer
List<Token> tokens
constructor: Parser(List<Token> tokens)
List<NoteElement> parseNotes(): returns a Piece Object with the noteElements based on tokens
class Token
enum type
String data
class KeyToAccidentals
getAccidentals(String key): returns a HashMap<Character, String> representing the accidentals in the input key
class PlayableNotes
This is a class with attributes that are needed for creating a pitch and adding a sound to the sequence player. A playableNote can only be created from a note.