6.005 Project 1 Final Design

TA: Leonid Grinberg

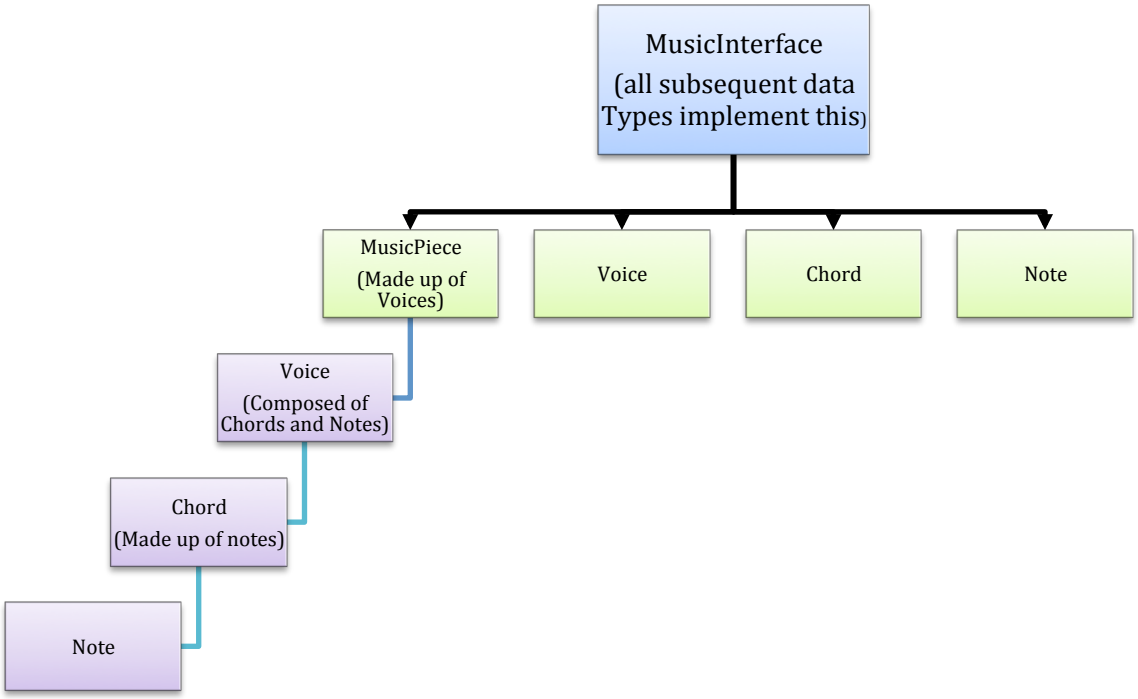
Team
Vladimir Bok : vbok@mit.edu
George Pantazis: geopant@mit.edu
Henry Nassif: hnassif@mit.edu

Outline

The high level organization of our project is the following. We process the abc music file into two strings -- one for the header and one for the body. We then feed the resulting strings into header and body lexers, respectively, which then return a sequence of tokens to the parser. The Parser handles the tokens and creates abstract data structures representing musical objects that are of one of the following types: Note, Chord, Voice, Song, each an implementation of a single Music interface. Song consists of a list of Voices where each voice consists of a list of Chords, where each Chord consists of a list of notes. (Note: Tuplet is handled simply as a number of notes in succession. There is no special class for it.)

To process the parser output, we use the visitor design pattern. The Music Visitor visits the instance of the Song object from the parser and returns a List<AugmentedMIDI>, ready to be fed into the **Java MIDI Sequence player**. This completes the general description of our design.

The diagram below shows the general structure of our program:



Lexer

The Lexer is passed a String representation of the entire abc music file. It then proceeds to split the string into a headerString, a string representing the header, and bodyString, a string containing the body. To avoid any confusion between header and body of the abc string, the lexer makes the distinction early on and stores each one in a different field, tokenizing each one according to their own grammar. To differentiate between the header and body, we perform a regex match to find the end of the Key Signature field in the header, “K:”. Anything after this line is considered part of the body and anything before is part of the header.

We opted to proceed with using Tokens to process the body and a Map<Character, List<String>> to process the header. The Character (the Key in this Map) represents the “label” of the given header field and the List (the value associated with a given Key in the Map) stores the value of the given header field. We chose to use a list as the map value field so that we can have Multimap functionality, enabling us to store multiple voices under the same key, ‘V’.

For the Body Lexer, we tokenize by having a regular expression matcher which yields each of the possible tokens that can appear (as enumerated in the BodyToken class) and their associated value fields (stored as Strings). The Body Lexer stores its position in the bodyString and moves along it (forward or backward) according to the calls made to it by the Parser.

We use of the following methods in our implementation:

Lexer.Lexer():
The lexer’s constructor uses the given specifications to determine the start and end of the different parts of the input. The main Key for the distinction will be the string “K:” that is not part of a comment (i.e. not preceded by “%”) to determine the end of the header and start of the body. This field is found using regex matching. It then proceeds to create instances of a headerLexer and a bodyLexer.

Lexer.getHeadLexer():
Method that returns an instance of headerLexer created by the constructor.

Lexer.getNextBodyToken():
Method to iterate through the body and return BodyTokens. Exists so it is simpler for the parser to obtain these tokens. Internally it calls the getNextBodyToken() method inside the BodyLexer itself.

Lexer.goBack():
Method used to handle repeats. It is defined in the Lexer for the ease of the parser. Internally, it calls the same method found in the BodyLexer. It moves the

lexer back to the start of a repeat segment. This method is called by the parser, whenever a token indicating the end of repeat segment is encountered. Works in conjunction with the Delete(), a method to remove tokens not needed anymore after repeat (see below).

Lexer.Delete():
Method in the Lexer to facilitate the implementation of the parser. Internally, the Lexer calls the equivalent method in its BodyLexer field. This method is called by the parser whenever some repeated tokens need to be discarded after the first repetition. Works in conjunction with goBack() so as to handle repeats which require ignoring a part of a piece of music.

The main methods in Header and Body Lexers are listed below, including a short description.

HeadLexer

HeadLexer()
Lexes the Header by populating a hash map where the keys are the header fields and the value is the field's value.

BodyLexer

BodyLexer.BodyLexer()
Initializes the field String body which holds the body of the abc music file. Moreover, it creates a regex matcher over said string and also creates an empty ArrayList<Integer> which holds the locations visited so far in the body (index in the string). This ArrayList<Integer> will be useful for our rewinding methods.

The regexes and tokens used for Lexing the body are as follows:

Voice:	"(V:[^\\n]+)|"
Comment:	"(%[^\\n]*)|"
Extra Repeat:	"(\\[[12])|"
Accidental:	"([\\^_=]+)|"
Note or Rest:	"([a-gA-Gz])|"
Multiplicative Factor:	"([1-9]*/[1-9]*|[1-9]+)|"
Octave Modifier:	"([\',]+)|"
Section Begin:	"(\\[\\\|)|"
Section End:	"(\\\|])|"
Repeat Begin:	"(\\\|:)|"
Repeat End:	"(:\\\|)|"
Begin Chord:	"(\\[)|"
End Chord:	"(])|"

Begin Tuplet:	"(\\()|"
Bar:	"(\\\|)"

BodyLexer.getNextBodyToken():
This method is called from the Lexer itself. It proceeds through the body string from the current location to the end and skips all unmatched sequences until it finds a match with a token or reaches the end of a file, at which point it returns an EOF Token (indication “end of file”).

BodyLexer.goBackUntil():
This method takes a string argument, “wanted”, to which we want to rewind. It rewinds the bodyLexer until it finds the last occurrence of the desired string. Makes use of a helper function, lastIndexOf() from the String class, to get the Last index at which we encountered the wanted string. If the string exists in the bodyString, the currentLocation is set to the index of Last occurrence of the wanted string and returns a boolean True. Otherwise, a boolean false is returned.

BodyLexer.goBack():
Similarly to the previous method only this method rewinds the currentLocationIndex to the last major Section encountered, which is denoted by either a voice declaration, “V:” or a repeat start, “|:”, or finally a section begin, “[|”.

BodyLexer.deleteLastToken():
It deletes the last BodyToken seen so far and proceeds to modify the bodyString to reflect this deletion, by removing the appropriate elements from the String. It also places our currentLocationIndex to just before the start of the previous, now deleted, token.

BodyLexer.peek():
This method is similar to getNextBodyToken() in that it returns the next token found through regex matching, but unlike its counterpart it does not modify the currentLocation of the Lexer. It is functionally analogous to a Stack’s peek method, in that it looks at the following element without changing its state. This is useful for lookAhead in the parser.

Parser

The parser returns an abstracted version of the input: an instance of the Song object, which consists of a list of Voices where each voice consists of a list of Chords, where each Chord consists of a list of Notes. We use a Chord to represent all musical note types, that is proper chords, single notes and tuplets by modifying the speed, tempo and make up of each Chord in a Voice.

The Parser class does not parse the header directly. The output of the header lexer is processed in the constructor of the Song class, which is called in Parser.

The constructor either fills the header fields with the proper values based on the lexed output or uses one of the default values as specified in the ABC Subset.

The Parser processes the body. We use the “state machine” design such that the Parser retrieves body Tokens one by one and processes them sequentially. We use a number of helper methods to process every constituent unit of an abc music file that consists of multiple tokens, such as notes with accidentals. Each of these methods returns an instance of the Note object. The parser collates the Notes into Chords and Chords into Voices, which are used to create an instance of Song, which the parser outputs.

Method
The Parser applies modifiers to notes, taking into account both modifiers mentioned immediately before the note as well as modifiers mentioned in the Key signature of the piece. The MusicalVisitor then visits the abstract data structure produced by the Parser and returns a List of augmentedMIDI, which is fed into the sequence player in the main method.

Below is a list of important “helper” classes the parser utilizez, along with certain critical fields. These allow the Parser to properly parse abc file.

Modifiers Class
We have a Modifiers class that holds a “Modifiers” Map<Pitch, Pitch> between the original pitch and its modified version. This map is initially built from the default modifiers mentioned in the Key signature field in the file header. As the parser parses through the body, the applyModifier method is called on each token of type note. This method applies the modifier to the note as well as keeps track of the modifier on the note in the modifiers map, so that it is applied to any similar note within the same bar. At the end of the current bar and the beginning of the next bar, the Modifiers map is reinitialized to its default value and the effect of the key signature is restored.

Duration Field
The duration of any Note, Chord, Song, Voice will be represented by a Multiplicative Factor: A ratio of two integers in its simplest form. We abstract the duration into two integers, one representing the denominator, and the other one representing the numerator, both of which are private fields in the MultiplicativeFactor class.

Multiplicative Factor
This Auxiliary class holds addition, subtraction, multiplication, compare, and toString methods to help facilitate operations between every instance. It is used to represent both the length of, Notes, Chords, Voices and the entire song, and the multiplicative factor modifiers applied after Notes.

AugmentedMIDI
A MIDI note augmented with all the necessary information for it to be correctly played by a sequence player. The fields are duration, pitch, and start. This class contains all the information needed to play a Note within one instance.

MusicInterface
This interface holds the declaration for all methods shared by Note, Chord, Song, Voice, such as the accept method for the MusicalVisitor and the getLength() method. Musical visitor is itself an interface nested within MusicInterface, and contains the methods to be used by MusicVisitor.

MusicVisitor
This class holds the body for the different visit method applied on each of Note, Chord, Song, and Voice. These methods are called by dynamic dispatch after the accept method of the corresponding object is called. The methods return a list of AugmentedMIDI notes ready to be played and with all necessary modifications (pitch, duration, octave, etc.) applied to them.

In more detail, the important fields and methods in each helper class are outlined below:

Interface MusicInterface:
accept(MusicalVisitor) --//Accepts the visitor
getLength(Musical Obj) --//Returns the length of the musical obj and uses dynamic dispatch to call the right getLength() function for each object (Song, Voice, Chord, Note)

class Note implements MusicInterface:
Private Pitch pitch; // The pitch of a note
Private MultiplicativeFactor ratio // used to represent the length of a note as a ratio

class Chord implements MusicInterface:
Private List<Note> // The list of notes that make up this chord. A chord can specify either a single note, a tuplet and of course a proper chord, based on the note list found inside it.
Private MultiplicativeFactor duration // the duration of the entire chord

class Voice implements MusicInterface
Private List<Chord> // All the “Chords” found within a voice.
Private MultiplicativeFactor duration // the duration of this entire voice, which is the sum of durations of each of the individual “Chords” inside it.

class Song implements MusicInterface:
This class holds all the necessary fields to fully describe a song.

The map, VoiceMap is where each Voice (made up of the list of chords) is held. This map, along with the String description of the Header Fields, the tempo, and Length of the song (defined as the Length of the longest voice) fully describe the entire song which can then be built using AugmentedMIDI notes.

```
private String songTitle
private String composerName
private String trackNumber
private String key
private String meter

private MultiplicativeFactor defaultNoteLength
private int ticksForNote
private int tempo
private Map<String,Voice> voiceMap
private MultiplicativeFactor length
```

Testing

We test the following components of our system:

1. HeadLexer
2. BodyLexer
3. Parser
4. MultiplicativeFactor
5. Main (Entire System)

We have selected these components as this is where the majority of the work is done in our design. Moreover we expect most bugs to originate either in the Lexer or the Parser or in the interface between the two. More specifically:

HeadLexer
In our test suite for HeadLexer, we check that the header meets all the requirements as specified in the ABC Subset for 6.005, such as the proper ordering of the header fields.

BodyLexer
Here we test that we can tokenize a variety of different inputs correctly. We aim to have tests to explore each of the individual possible inputs, such as Notes, Chords, Tuplets, and multiple Voices. Moreover we include tests that incorporate sequences of the above (notes, Chords, etc) and importantly tests for Repeats (this is not critical as much in the Lexer as is for the Parser). We also test invalid abc files and make sure meaningful exceptions are thrown in each case.

Parser
The tests here are to ensure that the Parser correctly builds the ADT to represent the Song by properly identifying the tokens and applying all necessary Accidentals to individual Notes. As in the Lexer, we again aim to implement a full range of tests to explore a variety of possible inputs. An important test will be to examine whether Repeats are handled correctly as are multiple Voices.

Multiplicative Factor
Here we test whether operations can be correctly performed on multiple instances of the class and that the representation invariant is always maintained : i.e.: the fraction is always in its simplest form.

Main
This is the final test suite we implemented; it is designed to ensure that inputs correctly propagate through our entire system and produce the correct outputs. This allows us to test the overall functionality of our system and identify any lingering bugs. In addition, it helps us check for correctness. We make sure to test a wide variety of abc files to ensure the correctness of our program.