**ABC Player Design Notes**

Catherine Zuo, Kimberly Toy, Will Oursler

**Contents**

**The following document addresses the following:**

1. Dependency Diagram
2. Design Flow, includes explanation of how the parser will work and explanation of the representation of the input and how it will be transformed into a playable form.
3. List of Classes and Data types, with descriptions of methods and whether it is mutable or not.
4. As we do not wish to modify the given grammar, we will not be including notes on the design of our new grammar
5. Explanation of testing

* Think about the datatypes your music player will require. List these datatypes, state whether they are mutable or immutable, and list their operations. For each operation, provide a short description of what it does.
* Demonstrate how your parser is going to work. You can do this by describing its individual methods, by displaying a state machine diagram, or through some other format. Additionally, demonstrate how your parser will detect and handle semantic errors in an input abc file.
* Describe how you will take a representation of the input (e.g., your AST) and transform it into a format that you can cleanly play using SequencePlayer.
* List the components of your system that you believe can and should be tested. For each of these components, describe your testing strategy and describe at least three specific test cases you plan to have.

**1. Dependency Diagram**

The following dependency diagram describes all of the classes that will be included in the ABC music player. The boxes represent classes, which form common functionalities.

**Design Flow**

**Lexing**

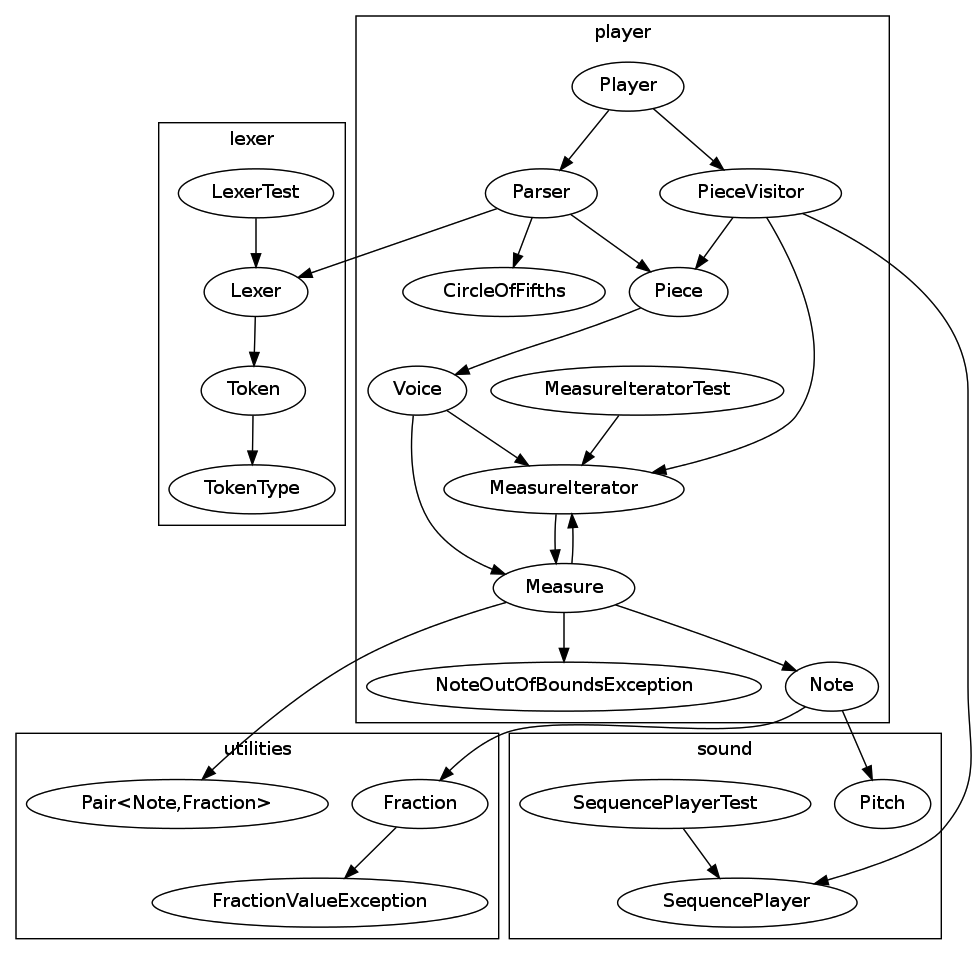
TokenType objects describe the type of tokens that can be made (i.e. note or accidental tokens) while Token objects represent the different kinds of tokens that are lexed from the .abc file contents.

A Lexer object is created with a list of TokenTypes representing the kinds of tokens that it can lex. It accepts the .abc text input and returns a list of valid tokens or throws an exception if invalid tokens are found.

**Parsing**

The Parser users the Lexer object to get the tokenized version of the music file and to create a Piece object, which is our representation of the .abc input.

The parser will use recursive descent parsing. It will look for each non-terminal, according to the given grammar

****

**List of Classes and Data types**

The following list shows all the classes that will be implemented for the ABC music player. They are grouped by functionality as shown by the boxes in the above dependency diagram. For the data type classes that describe the music

**Lexer Classes**

**public** **class** Lexer {

**private** **final** List<TokenType> types;

**public** Lexer(List<TokenType> types)

//Lexes the passed string into Tokens, returns a list of tokens

**public** List<Token> lex(String input) **throws** RuntimeException

}

**public** **class** Token {

**public** **final** TokenType type;

**public** **final** String contents;

**public** Token(String contents, TokenType type)

**public** String toString()

}

**public** **class** TokenType {

**public** **final** String name;

**public** **final** Pattern pattern;

**public** TokenType(String name, Pattern pattern)

}

**public** **class** LexerTest {

//Test file

}

**public** **class** CircleOfFifths {

//return the key signature for the given key

**public** getKeySignature(Note n)

}

**public** **class** Measure **implements** Iterable<Measure> {

//Length of the measure.

**private** **final** Fraction length;

//A list of notes, each associated with their start times

**private** List<Pair<Note, Fraction>> notes;

//The typical next measure in the larger piece.

**private** Measure next;

// An alternate next measure, e.g. the escape from a repeat.

**private** Measure alternateNext = **null**;

//Full constructor for Measure

**public** Measure(Measure next, Measure alternateNext,

List<Pair<Note, Fraction>> notes, Fraction length) **throws** NoteOutOfBoundsException

//Constructor with a list of empty notes

**public** Measure(Measure next, Measure alternateNext, Fraction length)

//Constructs measure only based off the next Measure

**public** Measure(Measure next, Fraction length)

//Default constructor

**public** Measure( Fraction length )

//Returns an iterator which will start with this measure, and continue until the end of the piece is reached.

**public** Iterator<Measure> iterator()

//Getter for this.next

**public** Measure getNext()

//Setter for this.next

**public** **void** setNext(Measure next)

//Getter for this.alternateNext

**public** Measure getAlternateNext()

//Setter for this.alternateNext

**public** **void** setAlternateNext(Measure alternateNext)

//Getter for this.notes

**public** List<Pair<Note, Fraction>> getNotes()

// Safe method to add a note to this measure.

**public** **void** addNote(Note note, Fraction startTime ) **throws** NoteOutOfBoundsException

}

**public** **class** MeasureIterator **implements** Iterator<Measure> {

//The measure we are currently initialized with.

Measure start;

//The measure we are currently concerned with.

Measure current;

//A Map containing how many times we have seen each measure by reference.

Map<Measure, Integer> timesSeen;

**public** MeasureIterator(Measure start)

//Check if there’s a next measure.

**public** **boolean** hasNext()

//Return the next measure.

**public** Measure next()

//This method always throws an error – prevents removing measures.

**public** **void** remove() **throws** RuntimeException

}

**public** **class** Note {

**public** **final** Fraction duration;

**public** **final** Pitch pitch;

**public** Note(Fraction duration, Pitch pitch)

}

**public** **class** Parser {

//Parse the given string as a Piece.

**public** **static** Piece parse( String abcContents )

}

**public** **class** Piece {

//The title of the piece.

**private** String title;

//The composer of the piece.

**private** String composer;

//Track Number of the piece.

**private** **int** trackNumber;

//Default length or duration of a note.

**private** Fraction defaultNoteLength;

//It determines the sum of the durations of all notes within a measure

**private** Fraction meter;

// The number of default-length notes per minute.

**private** **int** tempo;

//Determines the key signature for the piece.

**private** String key;

//The List of all the starting measures for each voice.

**private** List<Voice> voices;

//The (largest, ideally) smallest division needed such that the length of each note (and rest) is an integer multiple.

**private** Fraction smallestDivision;

// get the meter for the piece

**public** Fraction getMeter()

// set the meter for the piece

**public** **void** setMeter(Fraction meter)

// get the tempo for the piece

**public** **int** getTempo()

// set the tempo for the piece

**public** **void** setTempo(**int** tempo)

// get the smallest note division for the piece

**public** Fraction getSmallestDivision()

// set the smallest note division for the piece

**public** **void** setSmallestDivision(Fraction smallestDivision)

// get the key for the piece

**public** String getKey()

// set the key for the piece

**public** **void** setKey(String key)

// get the title for the piece

**public** String getTitle()

// set the title for the piece

**public** **void** setTitle(String title)

// get the composer name for the piece

**public** String getComposer()

// set the composer name for the piece

**public** **void** setComposer(String composer)

// get the track number for the piece

**public** **int** getTrackNumber()

// set the track number for the piece

**public** **void** setTrackNumber(**int** trackNumber)

// get the default note length for the piece

**public** Fraction getDefaultNoteLength()

// set the default note length for the piece

**public** **void** setDefaultNoteLength(Fraction defaultNoteLength)

// get the voices of this Piece

**public** List<Voice> getVoices()

// set the voices of this Piece

**public** **void** setVoices(List<Voice> voices)

}

**public** **class** PieceVisitor {

// Processes a Piece to play

**public** **static** SequencePlayer process(Piece piece)

**throws** MidiUnavailableException, InvalidMidiDataException

// Convert time (in fractional length) to ticks

**private** **static** **int** fractionToTicks(Fraction time, Fraction divisionLength)

}

**public** **class** Voice **implements** Iterable<Measure> {

**public** **final** String name;

**private** Measure firstMeasure;

**public** Voice(String name, Measure firstMeasure)

// Get starting measure of this voice

**public** Measure getStart()

// Get last measure of this voice

**public** Measure getEnd()

// Get a measure iterator, starting from this voice’s start measure

**public** Iterator<Measure> iterator()

}

// Represenation of a fraction

**public** **class** Fraction {

**public** **final** **int** numerator;

**public** **final** **int** denominator;

**public** Fraction(**int** value)

**public** Fraction(**int** numerator, **int** denominator)

// Finds gcd of two integers

**public** **static** **int** gcd(**int** first, **int** second)

// Finds lcm of two integers

**public** **static** **int** lcm(**int** first, **int** second)

// Finds sum of this plus a Fraction

**public** Fraction plus(Fraction other)

// Finds sum of this plus an int

**public** Fraction plus(**int** other)

// Finds difference of this and a Fraction

**public** Fraction minus(Fraction other)

// Finds difference of this and an int

**public** Fraction minus(**int** other)

// Finds product of this and a Fraction

**public** Fraction times(Fraction other)

// Finds product of this and an int

**public** Fraction times(**int** other)

// Finds quotient of this and a Fraction

**public** Fraction quotient(Fraction other)

// Finds quotient of this and an int

**public** Fraction quotient(**int** other)

// Finds the inverse of this

**public** Fraction inverse() **throws** FractionValueException

// Finds float approximation of this

**public** **float** approximation()

// Finds hash of this

**public** **int** hashCode()

// Equals method for Fraction class

**public** **boolean** equals(Object other)

// Express this Fraction as a String

**public** String toString()

}

// Custom exception for handling errors dealing with fractions

**public** **class** FractionValueException **extends** IllegalArgumentException {

**public** FractionValueException(String message)

}

// Class representing a Pair of generic types

**public** **class** Pair<First, Second> {

**public** First first;

**public** Second second;

**public** Pair(First first, Second second)

// Finds hash of this

**public** **int** hashCode()

// Equals method for Pair objects.

**public** **boolean** equals(Object other)

}