ABCPlayer: Milestone Design

Alvaro Morales, Eleftherios Ioannidis, Mariam Kobiashvili
{alvarom, elefthei, mari}@mit.edu

Tokens

Token is a class that has an enum with a set of values:

Type:

- HEADER_NUMBER
- HEADER_TITLE
- HEADER_COMPOSER
- HEADER_DEFAULT_LENGTH
- HEADER_METER
- HEADER_TEMPO
- HEADER_VOICE
- HEADER_KEY
- ACCIDENTAL
- BASENOTE
- OCTAVE
- REST
- NOTE_LENGTH
- CHORD_START
- CHORD_END
- DUPLET_START - “(2”
- TRIPLET_START - “(3”
- QUAD_START - “(4”
- BAR
- DOUBLE_BAR
- START_REPEAT
- END_REPEAT
- NTH_REPEAT_1
- NTH_REPEAT_2
- VOICE

The attributes of Token will be:

1. Type type

2. String value

We have two main groups of tokens:

- symbol tokens (no value): header field, start of a repeat, start of a chord, bar, etc.
- value tokens: basenote, accidental, octave, etc.

```
class Token
      Type type          // one of token types listed above, defined as enum
      String value
      Type getType()
      Integer getValue()
```

Lexer

The Lexer is constructed from a string input (the .abc file). It tokenizes the input, one token at a time using a PushbackBuffer.

For converting a substring of the input into a token, the Lexer class will have regular expressions for each terminal in the grammar. The grammar specifies the sequential order that these tokens should appear. For example,

```
      pitch ::= [accidental] basenote [octave]
```

If we see tokens in illegal orders, the Lexer will throw an unchecked LexerException. If we see illegal characters, we will throw a LexerException.

The Lexer will have a method next(), that produces the next token from the input. The Parser will consume these tokens.

Data Structures

```
// Used to represent durations of musical notes such as ¼ or ⅛
// Immutable

class RationalNumber
      Integer numerator;
      Integer denominator;
      Integer getNumerator();
      Integer getDenominator();
      Double getValue();
```

AST Definition

We will represent an abstract syntax tree (AST) for a song by the following recursive datatype.

Interfaces

```
// top-level interface for representing all note elements
// of a musical piece

interface NoteElement
      E accept(); // visitor pattern

// Used to represent tuples
// Can be one of duplet, triplet or quadruplet

interface Tuple
```

Classes

```
// Represents a single musical note
// Immutable

class SingleNote implements NoteElement
      char pitch //[A-G]
      RationalNumber duration
      int accidental //e.g. sharp = +1, flat = -1
      char getPitch()
      RationalNumber getDuration();
      int getAccidental()
      E accept() //visitor pattern



// Represents a rest
// Immutable

class Rest implements NoteElement
      RationalNumber duration
      RationalNumber getDuration()
      E accept() //visitor pattern


//Represents a chord
//Immutable

class Chord implements NoteElement
      RationalNumber duration
```

```
      ArrayList<NoteElement> notes
      RationalNumber getDuration()
      ArrayList<NoteElement> getNotes()
      E accept() //visitor pattern


// Represents a duplet
// Immutable

class Duplet implements NoteElement, Tuple
      NoteElement first
      NoteElement second
      NoteElement getFirstNoteElement()
      NoteElement getSecondNoteElement()
      RationalNumber getDuration() //plays 2 notes in the duration of 3
      E accept()                   // visitor pattern


// Represents a triplet
// Immutable

class Triplet implements NoteElement, Tuple
      NoteElement first
      NoteElement second
      NoteElement third
      NoteElement getFirstNoteElement()
      NoteElement getSecondNoteElement()
      NoteElement getThirdNoteElement()
      RationalNumber getDuration() //plays 3 notes in the time of 2
      E accept()                   //visitor pattern



// Represents a quadruplet
// Immutable

class Quadruplet implements NoteElement, Tuple
      NoteElement first
      NoteElement second
      NoteElement third
      NoteElement fourth
      NoteElement getFirstNoteElement()
      NoteElement getSecondNoteElement()
      NoteElement getThirdNoteElement()
      NoteElement getFourthNoteElement()
      RationalNumber getDuration() //plays 4 notes in the time of 3 notes
      E accept()                   // visitor pattern
```

```
// Represents a voice
// Mutable

class Voice implements NoteElement
      ArrayList<NoteElement> notes
      ArrayList<NoteElement> getNotes()
      void addNoteElement()

// Represents a song
// Mutable

class Song implements NoteElement
      ArrayList<Voice> voices
      ArrayList<Voice> getVoices()
      void addVoice()
      E accept() // visitor pattern
```

Parser

The parser will take a stream of tokens as an input from the lexer and produce an AST that the player can reproduce as an output. It will handle the header, repeats, chords and tuples, adding every incoming token to the appropriate datatype. An important component of the parser is a globally visible environment hashmap, which will keep track of all the header variables and the accidentals for each note (ie: env[‘C’] = ‘#’ if the parser finds a sharp for C in the current track). The parser will recursively create nested data structures when it encounters a delimiter, like ‘(‘ for tuples and ‘[‘ for chords, the class hierarchy for these structures helps our task.

The parser does additional validation checks, like checks for same meter on every block between two vertical bars, in which case, if the block is the last block of the piece it pads it with rests, otherwise throws a Parser exception.

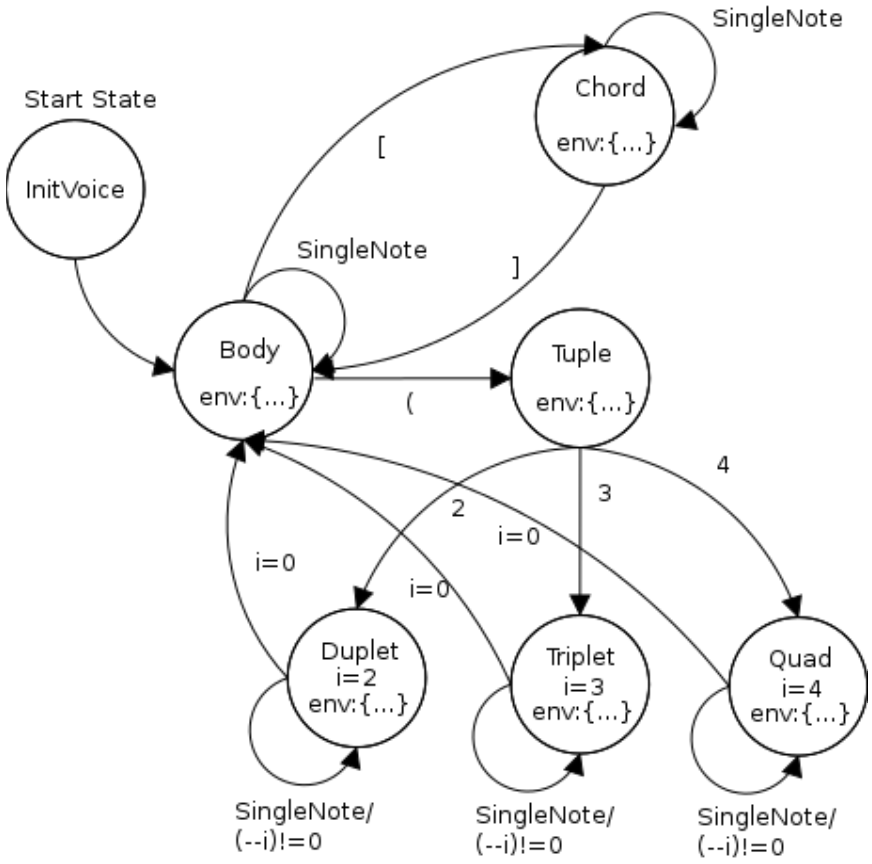
Header: The parser first starts with the header tokens coming from the lexer. It saves all the header information in the attributes of a new *song* object. After the parser gets a token for the key variable (i.e: K: C), it changes his state to *Body* and adds a new *voice* to the current *song,* named or unnamed, depending on the header information it has stored.

Now the parser is expecting the body of the abc file, ready to add *Notes* to the current *Voice*. If the parser encounters a header token (other than a voice change) it throws a *ParserException*, as we know the key is the last variable of the header.

Repeats: The parser handles repeats by keeping a token buffer for each voice and expecting a repeat symbol at the end of it (“:|“). This buffer is dynamically allocated at the beginning of each *voice*

definition. After the parser has met a repeat symbol, it pauses consuming tokens from the lexer and instead plays-back the buffer from his own memory. If a repeat symbol is not met, the buffer is discarded. Same goes for *nth-repeats* (i.e: "[1", "[2"), both the buffer and the starting repeat index are saved locally in the *voice* instance and are later used to playback the saved buffer.

Now that the non-regular parts of the abc notation are discussed, the rest (tuples, chords and rests) define a regular language, thus can be parsed by a state machine.



Player

The AST will be transformed to a playable format by the following classes:

```
// Used to implement the visitor pattern

interface Visitor
      E visit()         // visitor pattern, for each variant class

// Visits the AST
// Calculates the LCM of all the denominators to find
```

```
// the number of ticksPerQuarter

class DurationVisitor implements Visitor
      int ticksPerQuarter
      E visit()          // for each variant class
      int getTicksPerQuarter()

// Visits the AST
// Adds every NoteElement to the SequencePlayer by instantiating
// Pitch objects from the attributes of a NoteElement

class PlayerVisitor implements Visitor
      SequencePlayer player
      int currentTick
      E visit()          // for each variant class
      void advanceTick() // for each variant class
      SequencePlayer getPlayer()
```

We first visit the AST with a DurationVisitor. The job of the DurationVisitor is to figure out the number of ticks per quarter for the song. The visitor will look at the RationalNumber duration of each base NoteElement and calculate the lowest common multiple.

Once we know this information, we instantiate a PlayerVisitor with this parameter. This visitor will recursively add all the notes to the SequencePlayer in the environment.

To add a note to a SequencePlayer, we need to specify a Pitch instance, a startTick (start position of the note in the song) and a numTicks (duration of the note). For example:

```
player.addNote(new Pitch('C').toMidiNote(), 0, 1);
```

The numTicks can be found by multiplying the real value of a RationalNumber by the numTicksPerQuarter.

Adding a note means that the song should move forward (i.e. currentTick should be incremented). However, not every note advances the song in the same way. For example, notes in a chord play at the same time and the song should advance for a single duration. advanceTick() methods are defined for each variant class, so that the appropriate method is called depending on the type of the class. For every voice, currentTick is reset to 0.

Testing

Testing strategy for the Lexer

The LexerTest class should test that:

- Each individual token type is correctly lexed
- Whitespace is ignored
- Linefeeds are ignored and lexing continues after a newline character
- Comments are ignored
- Characters outside the grammar throw exceptions

Testing strategy for the Parser

The ParserTest class should test that:

- Header information is appropriately stored in a *Song*
- The Header contains the mandatory fields in the order specified by the grammar
- Tuples are parsed correctly
- Chords are parsed correctly
- Repeats are parsed correctly
- A single voice song is parsed correctly
- A multi-voice song is parsed correctly

Testing strategy for the Player

The PlayerTest class should test that:

- ticksPerQuarter is calculated correctly for songs with different components (chords, tuplets)
- a single voice song is played correctly
- a multiple voice song is played correctly