ABCPlayer Design

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

Overview

System Components:

1. Tokens
2. Lexer
3. Data Structures
4. AST Definition
5. Parser
6. Player

Directory Structure:

package **ast**:
    The classes that constitute our ABCPlayer's abstract syntax tree.
package **player**:
    The lexer, parser and player as well as their complementary data strucutes.
package **sound**:
    The API to the java midi interface.
package **test:**
    Our test suite, includes modular as well as integration JUnit tests.

* * *

Tokens

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

- INDEX
- TITLE
- COMPOSER
- LENGTH
- METER
- TEMPO

- VOICE
- KEY
- KEYNOTE
- REST
- CHORD_START
- CHORD_END
- DUPLET_START
- TRIPLET_START
- QUAD_START
- BAR
- DOUBLE_BAR
- REPEAT_START
- REPEAT_END
- REPEAT_NUMBER

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
      int octave
      int accidental
      RationalNumber Duration
```

All token types have appropriate getter and setter methods as well as a parseValue() method that takes a note in string form (i.e: A1/2) and assigns values to the right attributes (i.e: accidental=MAX_INT, Duration=1/2). Notes with no accidentals get an accidental=MAX_INT, as an accidental=0 means this note was preceded by a natural sign and we want to be able to distinguish between the two during parsing.

Lexer

The Lexer is constructed from a string input (the .abc file). It tokenizes the piece header and body separately using the methods `makeHead()`, `makeBody()`, `lexHead()` and `lexBody()` to separate body and head (using the Key header field K: as a delimiter) and lex body and head respectively.

For converting a substring of the input into a token, the Lexer class will have two monolithic regular expressions that match each and every one of the head and body terminals, for each terminal in the grammar and then specify which capturing group matched using the `matcher.group()` method in the java regex library. The grammar specifies the sequential order that these tokens should appear. For example,

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

The Lexer's `lexHead()` method returns an array list of header tokens and the `lexBody()` method returns an array list of body tokens.

The Lexer also implements an unchecked exception class called `LexerException`, which is thrown when:

1. The Key (K:) header field is not found so the lexer doesn't know how to split the head and body.
2. The Key (K:) header field is found but it's the last token of the input, meaning there is no body.
3. The number of characters in the header, excluding whitespace, is not the same as the number of characters matched by the header's regular expression. This means there are leftover characters after lexing the header.
4. The number of characters in the body, excluding whitespace, is not the same as the number of characters matched by the body's regular expression. This means there are leftover characters after lexing the body.

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
      RationalNumber getDuration()

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

interface Tuple
      RationalNumber getDuration()
```

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()
      boolean equals(NoteElement)
      ArrayList<NoteElement> getNotes()
      E accept() //visitor pattern
```

```
// Represents a duplet
// Immutable
class Duplet implements NoteElement, Tuple
      NoteElement first
      NoteElement second
      NoteElement getFirst()
      NoteElement getSecond()
      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 getFirst()
      NoteElement getSecond()
      NoteElement getThird()
      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 getFirst()
      NoteElement getSecond()
      NoteElement getThird()
      NoteElement getFourth()
      RationalNumber getDuration() //plays 4 notes in the time of 3 notes
      E accept()                   // visitor pattern

// Represents a voice
// Mutable

class Voice implements NoteElement
      String name
      ArrayList<NoteElement> notes
      String getName()
```

```
        ArrayList<NoteElement> getNotes()
        RationalNumber getDuration()
        void addNote()
        boolean equals(NoteElement)
        String toString()

// Represents a song
// Mutable

class Song implements NoteElement
        Map<String,Voice> voices
        accidentalAssociationMaker AccidentalAssociator
        String composer
        String title
        String keySignature
        int tempo
        int index
        RationalNumber defaultNoteLength
        RationalNumber meter
        Voice currentVoice
        void add()
        String getComposer()
        String getTitle()
        String getKeySignature()
        int getTempo()
        int getIndex()
        RationalNumber getMeter()
        Voice getVoice(String)
        List<Voice> getVoices()
        RationalNumber getDefaultNoteLength()

        void addVoice()
        void setComposer(String)
        void setTitle(String)
        void setKeySignature(String)
        void setTempo(int)
        void setIndex(int)
        void setMeter(RationalNumber)
        RationalNumber setDefaultDuration()

        boolean equals(NoteElement)
        String toString()
        E accept() // visitor pattern
```

Parser

The parser's parse() method takes two lists of tokens as input from the lexer and produces an AST of type `Song` that the player can reproduce. The parser will parse the header token list and the body token list separately. The parser will handle the header, repeats, chords and tuples, adding every instance to the appropriate datatype inside a Song.

An important component of the parser is a public environment hashmap, which is represented by the `AccidentalAssociationMaker` class. This `AccidentalAssociator` keeps track of the accidentals for each note pitch (ie: getAccidental('C') = '+1' if the parser finds a sharp for C in the current track). The AccidentalAssociator is initialized with the key signature of the song, stores temporary accidentals between bars and whenever a new bar is encountered, it reverts to the defaults for the song's key signature.

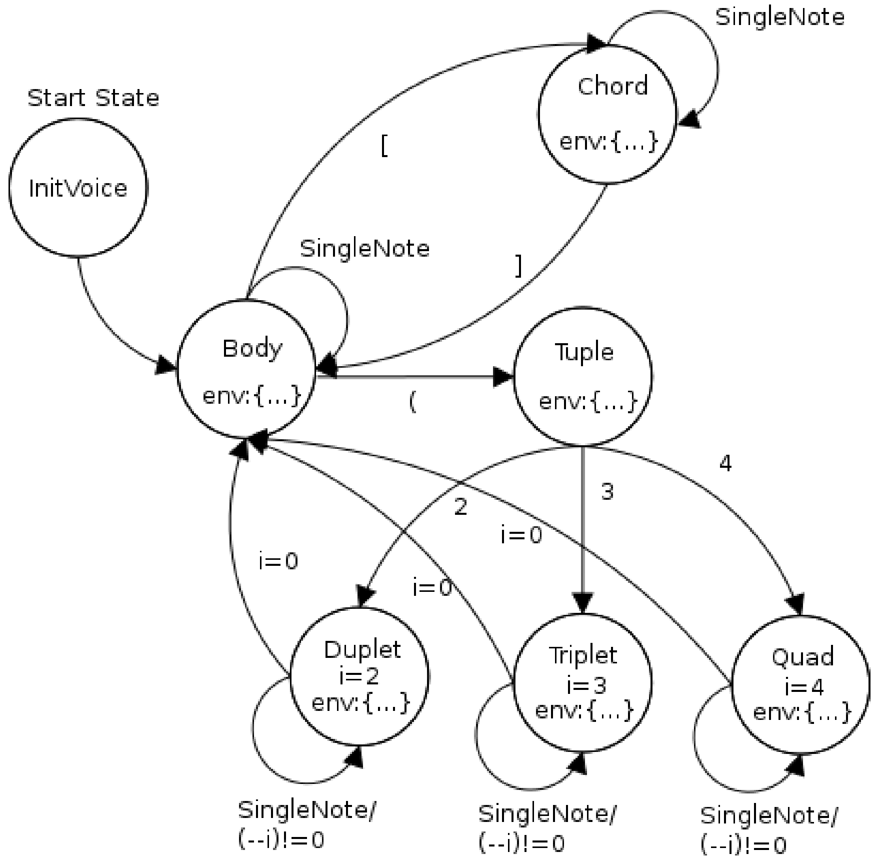
Header: The parser first calls the parseHeader() method with the header tokens list coming from the lexer. It saves all the header information in the attributes of a new *song* object. After the parser reaches the end of the header tokens list and gets a token for the key variable (i.e: K: C), it starts parsing the *Body* list and adds a new *voice* to the current *song,* named or unnamed, depending on the header information it has stored. The parser enforces header rules: the first token must be an "X:", followed by a "T:", optionally followed by more valid header tokens, and ending with "K:".

The parser then breaks up the list of body tokens by voice. This simplifies handling repeats broken up into different lines in the input file. Now the parser is ready to add *NoteElements* to the current *Voice*, which is described by the State Machine diagram at the end of the page.

Repeats: The parser handles repeats by keeping an index number for each voice, and every time a REPEAT_END token is parsed, it moves the token list iterator to that stored index, which is the index of the previous REPEAT_START token. It marks both REPEAT tokens as "PASS" and moves on. A REPEAT_START or REPEAT_END token marked as "PASS" is ignored.

Same goes for *nth-repeats* (i.e: "[1", "[2"), when an nth-repeat is met, the parser searches backwards for any delimiter (BAR, DOUBLE_BAR, REPEAT_END, REPEAT_START, REPEAT_NUMBER) and stores it's index inside the value and octave of the nth-repeat token. That way, it can move the token list iterator to that index after the current block has been completely parsed.

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.



The Parser also implements and throws his own exception class, `ParserException`, which is an unchecked exception that inherits from `RuntimeException`. A `ParserException` is thrown at the following cases:

1. A NoteElement which is not a SingleNote is found inside a chord. This is unspecified behavior, so an exception is thrown.
2. A DUPLET_START token is found but it isn't followed by at least two SingleNotes or Chords.
3. A TRIPLET_START token is found but it isn't followed by at least three SingleNotes or Chords.
4. A QUAD_START token is found but it isn't followed by at least four SingleNotes or Chords.
5. Header size is less than three tokens. As the header is required to have the X:, T: and K: fields set, this behavior is unspecified.
6. X: is not the first field in the header. As this is a requirement of the ABC 6.005 subset, an exception is necessary.
7. T: is not the second field in the header. As this is a requirement of the ABC 6.005 subset, an exception is necessary.
8. K: is not the last field in the header. As this is a requirement of the ABC 6.005 subset, an exception is necessary.
9. No K: field found in the header. As this is a requirement of the ABC 6.005 subset, an exception is

necessary.
10. A chord started but the parser couldn't find the end of it.
11. A header token is found in the body. This is prohibited in the ABC 6.005 subset.
12. An undeclared new voice appears in the middle of the body.

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.
- Extra header fields and body elements 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
- Parser exceptions are thrown when necessary

The ParserIntegrationTest class runs the same tests, but uses the Lexer to get the input tokens to the Parser, testing these two components together.

Testing strategy for the Player

The DurationVisitorTest 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

The PlayerTest class should test our 3 abc files:

- playing pokemon.abc should tests playing chords nested in tuples
- playing zelda.abc should test playing a song with multiple voices
- playing jack.abc should test numbered repeats

PlayerTest tests the all the system components working together (integration testing).

Changes

We implemented a lot of changes compared to our initial ABCPlayer Milestone Design, mostly due to make our implementation simpler and more minimalistic. One major change was migrating from a lexer that uses the iterator pattern to a lexer that outputs a full list of tokens. This allows the parser to loop inside the token list, move the token list iterator to handle repeats and nth-repeats and in general gives our parser implementation more freedom.

Another important change was moving from a buffered repeats design to a design using “smart” iterators that can point to different points in the token list that we want to repeat. This was partly due to our list-of-tokens change of heart, as now we could move freely inside the token list instead of being restrained in only one token at a time.

In our parser, we moved from an independent hashmap of note-to-accidental to a whole new class, with features like revert() and a huge constructor that can create an association hashmap out of any given major or minor key. The main reason we moved to a new class was the size of the code.

Another marginal case we handled our implementation was repeats across the same voice, despite a second voice appearing. We fixed that by making the repeat index applicable per voice instead of per song. We realized that to handle repeats in voices easily, it was important to pre-process the list of tokens and separate them by voice.

Last but not least, our implementation is capable of handling chords nested inside tuples, something we didn’t consider in our initial design. The parseTuple() method in the parser adds the appropriate number of NoteElements (not just SingleNotes, as we originally thought) to the Tuple.