Project 1 Design

Alex Chernyakhovsky, Charles Liu, Lauren Stephens

1 Grammar

```
abc-tune ::= abc-header abc-music

abc-header ::= field-number comment* field-title other-fields* field-key

field-number ::= "X:" DIGIT+ end-of-line
field-title ::= "T:" text end-of-line
other-fields ::= field-composer | field-default-length | field-meter
  | field-tempo | field-voice | comment
field-composer ::= "C:" text end-of-line
field-default-length ::= "L:" note-length-strict end-of-line
field-meter ::= "M:" meter end-of-line
field-tempo ::= "Q:" tempo end-of-line
field-voice ::= "V:" text end-of-line
field-key ::= "K:" key end-of-line

key ::= keynote [mode-minor]
keynote ::= basenote [key-accidental]
key-accidental ::= "#" | "b"
mode-minor ::= "m"

meter ::= "C" | "C|" | meter-fraction
meter-fraction ::= DIGIT+ "/" DIGIT+

tempo ::= DIGIT+

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

abc-music ::= abc-line+
abc-line ::= (element+ linefeed) | mid-tune-field | comment
element ::= note-element | tuplet-element | barline | nth-repeat | space
```

```
note-element ::= (note | multi-note)

// note is either a pitch or a rest
note ::= note-or-rest [note-length]
note-or-rest ::= pitch | rest
pitch ::= [accidental] basenote [octave]
octave ::= ("'"+) | (","+)
note-length ::= [DIGIT+] ["/" [DIGIT+]]
note-length-strict ::= DIGIT+ "/" DIGIT+

; "^" is sharp, "_" is flat, and "=" is neutral
accidental ::= "^" | "^^" | "_" | "__" | "="

basenote ::= "C" | "D" | "E" | "F" | "G" | "A" | "B"
        | "c" | "d" | "e" | "f" | "g" | "a" | "b"

rest ::= "z"

// tuplets
tuplet-element ::= tuplet-spec note-element+
tuplet-spec ::= "(" DIGIT

// chords
multi-note ::= "[" note+ "]"

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

; A voice field might reappear in the middle of a piece
; to indicate the change of a voice
mid-tune-field ::= field-voice

comment ::= "%" text linefeed
end-of-line ::= comment | linefeed
```

2 Data Types

2.1 Lexer

The first stage of reading the abc file is converting the text data to tokens. This requires the creation of a `Token` class, which will have two fields: type and an optional `String` data. The following types must exist:

- `FIELD`
  Each field will be given its own value in the enumeration, and the optional data will

be filled in. Specifically, these fields exist:

  - `FIELD_INDEX_NUMBER`
  - `FIELD_TITLE`
  - `FIELD_COMPOSER_NAME`
  - `FIELD_DEFAULT_LENGTH`
  - `FIELD_METER`
  - `FIELD_TEMPO`
  - `FIELD_KEY`
  - `FIELD_VOICE`

- `DIGIT`
  The digit token will store a single number in the optional data field.

- `NOTE_LETTER`
  A single note, which will store the actual pitch-letter in the optional data field.

- `ACCIDENTAL`
  A sharp or a flat, with the type stored in the optional data field.

- `BARLINE`
  Signifies the end of a measure, with the optional data field holding the variations.

- `NTH_REPEAT`
  Variations of the repeat will be stored in the optional data field.

- `MULTINOTE`
  Indicates a sequence of notes is a member of a chord, or that the chord is over. Specifically,

  - `BEGIN_MULTINOTE`
  - `END_MULTINOTE`

- `OCTAVE`
  Modifier for the note, either up or down.

- `FRACTION_BAR`
  A literal fraction bar, "/".

- `COMMENT`
  A comment.

- `TUPLET`
  Indicates the beginning of a tuplet, with the optional data containing the size of the tuplet.

The Lexer considers the following characters as whitespace: " ", "
t", "
n", "
r". All of these will be ignored.

2.2 Parser

These tokens will then be combined by the Parser into an Abstract Syntax Tree. The root of the tree will be the `Music` class, which has `Voice`s in it. Each `Voice` will contain `MusicSequence`s; a `MusicSequence` is either a `Bar` or `Repeat`. A `Repeat` will have 1 or more `Bar`s. `Bar`s will contain `Accidental`s and `Note`s.

$$
\begin{aligned}
Music &::= Map < String, Voice > \\
Voice &::= List < MusicSequence > \\
MusicSequence &::= Bar() + Repeat() \\
Repeat &::= List < Bar > \\
Bar &::= List < MusicalElement > \\
MusicalElementContainer &::= Bar() + Chord() + Tuplet() \\
ContainerType &::= ENUM(BAR, CHORD, TUPLET, REPEAT, VOICE) \\
MusicalElement &::= Accidental(pitch : Pitch, value : int) \\
&\quad + Note(pitch : Pitch, len : MusicalLength(num : int, denom : int) \\
&\quad + Chord() + Tuplet() \\
Chord &::= List < Note > \\
Tuplet &::= List < Note >
\end{aligned}
$$

3 Traversing the Abstract Syntax Tree

The AST will be traversed using the Visitor pattern. The Visitor will go through each level of the tree, applying correct transformations, until it reaches a `Note`, at which point the `Note` will be added to the SequencePlayer.

4 State Machines

4.1 Parser

The parser needs to maintain knowledge about the current parsing of repeats, so that it knows if the current bar is alone or a member of the repeat, or even ending the repeat.

4.2 TranslateToSequenceVisitor

This Visitor must maintain the state of the accidentals in the `Bar`. This state variable is cleared out whenever a new `Bar` is entered. Any errors in the AST will be played, with minor corrections applied (e.g., invalid chord lengths will be reduced to the length of the first note).

4.3 OptimalTicksPerQuarterNoteVisitor

This Visitor calculates the Least Common Multiple of the denominators of note length, thereby providing the optimal number of ticks.

4.4 MusicalWellFormednessVisitor

This Visitor checks the Abstract Syntax Tree and returns any errors that it finds. It does not attempt to fix any errors.

5 Snapshot Diagrams

Snapshot diagrams are too large to include in this document. Please see the diagram files in the same folder as this document.