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 ["m"]
keynote ::= basenote [key-accidental]
key-accidental ::= "#" | "b"
note-length-strict ::= DIGIT+ "/" DIGIT+

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

tempo ::= DIGIT+

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

abc-music ::= abc-line+
abc-line ::= (element+ linefeed) | field-voice | comment
playElement ::= note | chord | tuplet | rest

element ::= playElement | barline | nth-repeat

note ::= pitch [note-length]
pitch ::= [accidental] basenote [octave]
octave ::= ("'"+) | (","+)
note-length ::= [DIGIT+] ["/" [DIGIT+]]


accidental ::= "^" | "^^" | "_" | "__" | "="
```

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

rest ::= "z" [note-length]

// tuplets
tuplet ::= "(" DIGIT  [ note+ ]

// chords
chord ::= "[" note+ "]"

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

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

Tokens (Groupings of terminals) used for Lexing

Header Tokens

- Title
- Composer's name
- Meter
- Tempo
- Voice
- Key
- Index Number

Body Tokens

- Note
- Rest
- Chords
- Tuplets
- Repeats
- Barlines
- Change of voice token
- End of line

Design

We would have a lexer that converts all of our inputs to appropriate tokens, as listed above. We will use a specific token class to encapsulate such a Token object. It will have two fields a type and a value. The type will be an enum containing all the types of token seen above. Although we make the distinction between header tokens and body tokens we will not make two enums. Rather, the parser will have to determine when header tokens have finished and body tokens are beginning. The value will simply be a String representing the contents of the token. Although we could break down the values into pieces, we don't believe this is good design. The lexer should not have to handle the structure of the language at all. It should simply identify text. Therefore, the parser will be left to separate the String value into any necessary parts. Finally, it is important to note that the lexer will not construct all the tokens at once. Rather, it will convert one token at a time. This way, the parser can parse dynamically without all the lexing having to be done.

The next piece of the design is the parser. The parser will read the tokens one by one and put them together into an abstract syntax tree. Here are some details about the class structure and recursive datatypes we would use to accomplish this.

First of all, we would have a "Piece" class that has instance variables for all the header fields that the parser takes in. This would include the key, meter, tempo, etc. Next, we would have a voice class. This represents all the playable objects, rests, and everything within one voice.

Next, we would have a Playable interface that represents all musical objects that can be "played" (we include a rest here to make the design as clean and simple as possible). The classes Note, Rest, and Tuplet will all implement this interface. the Note class will encapsulate all the details relevant to a note - the pitch, the length, the accidentals, etc. The Rest class will have one field for the length. And finally, the Tuplet class will have a Set of notes and a type indicating what type of tuplet it is. Based on the specification in the 6.005 subset for abc, the number of notes in the tuplet should uniquely determine the way it is played. However, having a type (triplet, dublet, or quadruplet) will make things a little cleaner (Note: There is some ambiguity about how to handle tuplets. If needed, we will talk to the staff and change our design).

Now, here is how we will handle repeats. Essentially, this will be done by the parser. The parser will keep track of the most recent repeat signs it has seen and essentially unravel the repeats.

So in conclusion, here's what we'll have:

- Interface called Playable. We have classes Note, Rest, Chord and Tuplet that implement this interface.
- Overall Piece class, which has instance fields that include all data from headers.
- Each Piece class has Voice object(s). Voice object keeps track of time in each voice, notes to be played by that voice, etc. More specifically, the Voice class will probably just maintain a List of Playable objects.

- Each Piece object has a play method that returns a SequencePlayer that can be played.
    - NOTE: This will not involve multithreading at all. Notes that are played at the same time will just have the same tick values in the SequencePlayer.