**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. Also, we encourage (but do not require) you to include a dependency diagram.**

- **ReadyToAddNote**: directly stores the information needed to add a single note to the SequencePlayer
    - Immutable
    - Fields
        - midiNote (int)
        - startTick (int)
        - numTicks (int)
- **Duration**: represents a duration in terms of musical length (e.g. "quarter note" would be ¼)
    - Immutable
    - Fields
        - numerator (int)
        - denominator (int)
    - Operations
        - multiply [producer] - multiplies by a given fraction, returns a new Duration object
    - stored as a fraction in simplest terms
        - rep invariant: gcd(numerator, denominator) = 1
- **Pitch**: represents a musical pitch
    - Immutable
    - (this class is already given to us, so we won't describe it here)
- **Note**: represents a musical note
    - Immutable
    - Fields
        - duration (Duration)
        - pitch (Pitch)
        - accidental - describes whether there is an accidental attached to this note, and if so, what it is
    - Operations
        - multiply [producer] - multiplies the duration by a given fraction
- **Chord** [implements ChordSequence]: represents a note, chord, or rest
    - Immutable
    - Fields
        - notes (List<Note>)
            - if length = 0, then it's a rest
            - if length = 1, then it's a single note
            - if length > 1, then it's an "actual chord"
        - duration (Duration)
            - the duration of the chord as a whole (this is the duration of the first note in the chord as specified in the abc format)
        - syllable (String)

                - the text aligned with this Chord (if any)
    - Operations
        - getChords() [observer] - see ChordSequence interface - returns a singleton list containing this chord since calling it a "sequence"
        - multiply [producer] - multiplies the duration (and duration of all contained notes) by a given fraction
- **ChordSequence** [interface]: this represents any contiguous block of music from a single voice, including lyrics (because Chords contain their associated syllables)
    - not immutable/mutable because it's an interface
    - Operations
        - getChords() [observer] - returns a List<Chord> that represents a contiguous block of music
            - each Chord starts when the previous one ends, as encoded by Chord.duration (not to be confused with the duration of the individual notes within each Chord)
- **Tuplet** [implements ChordSequence]: represents a tuplet
    - Immutable
    - Fields
        - type (TupleType enum) - duplet, triplet, or quadruplet
        - chords (List<Chord>) - the 2, 3, or 4 chords that make up this tuplet
    - Operations
        - getChords() [observer] - see ChordSequence interface - returns a list of the chords this tuplet contains, but scales their durations appropriately (based on the tuplet type)
- **Measure** [implements ChordSequence]: represents a measure and all the music inside it, plus the key signature and accidentals
    - Immutable
    - Fields
        - chordSequences (List<ChordSequence>) - the ChordSequences (in this case Chords and Tuplets) that make up this measure
        - key (dictionary from musical notes to sharps/flats) - the key for this piece, used when applying sharps/flats in getChords()
    - Operations
        - getChords() [observer] - see ChordSequence interface - returns a list of the chords this measure contains (recursively calls getChords() on each ChordSequence in the measure), applies the key signature and accidentals to the notes in the returned list, and strips the accidentals off the notes in that list
- **BranchingRepeat** [implements ChordSequence]: represents a section of music with a repeat, but alternate endings
    - Immutable
    - Fields
        - common (ChordSequence) - the beginning (that's played every time)
        - endings (List<ChordSequence>) - a list of the possible endings
    - Operations

        - getChords() [observer] - see ChordSequence interface - calls getChords() on the contained chord sequence and then returns a list that's common + endings[0] + common + endings[1] + … (essentially unrolling the loop)
- **SimpleRepeat** [implements ChordSequence]: represents a section of music that repeats itself exactly
    - Immutable
    - Fields
        - repeat (BranchingRepeat)
            - common: the section to repeat
            - endings: a list of length 2 with an empty chordsequence in each slot
    - Operations
        - getChords() [observer] - see ChordSequence interface - calls getChords() on the contained BranchingRepeat, which has the effect of returning repeat.common + repeat.common as desired
- **Voice** [implements ChordSequence]: represents a voice (this is the highest level of ChordSequence)
    - Immutable
    - Fields
        - chordSequences (List<ChordSequence>) - a list of all the chord sequences that make up this voice
    - Operations
        - getChords() [observer] - see ChordSequence interface - returns a list of the chords this voice contains (recursively calls getChords() on each ChordSequence in the voice and concatenates them)
- **Song**: represents an entire song
    - Immutable
    - Fields
        - voices (List<Voice>)
        - key
        - meter/tempo
    - Operations
        - addToSequencePlayer - calls getChords() on each voice, computes the appropriate number of ticks per beat by using the LCM of the denominators of each duration, computes the starting ticks for each note of each chord, and adds them to the sequence player (using ReadyToAddNotes as an intermediary) (also does this for lyrics in each chord)

**You may wish to modify the given grammar we give you to make parsing input easier. This is ok, as long as it continues to fit the specifications. If you do wish to modify the grammar, include your modified grammar in your design.**

- We'll split the given BNF grammar into a header and content section which we'll use with ANTLR.

**Describe your strategy for using ANTLR to create your AST. You should also describe how you**

**will handle errors in the input file.**

- We'll convert the BNF grammars for header and footer into ANTLR grammar
    - We'll define things like DIGIT which aren't explicitly defined in the grammar, but are used in it
    - We'll use uppercase for lexical rules, lowercase for parser rules
- We'll send the proper .abc file through the converted ANTLR grammar and the generated lexer and parser using ANTLRFileStream(String fileName)
    - We will log errors to the console, but attempt to continue and play with the caveat that the resulting audio may sound nothing like what was intended.

**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.**

- Once we have an AST, we'll create a custom ParseTreeListener which will crawl through the tree, generating Notes (and their Pitches and Durations), Chords (and their List of Notes), Measures, Lyrics and Voices as it exits the various nodes of the tree. We'll then wrap it in a Song to combine the header metadata and the Voices.
- These will be played using the ChordSequence datatype's getChords() method, which is passed through a method to convert the notes into ReadyToAddNotes, whose arguments are then fed into the 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.**

- The three components of our system are the lexer, the parser, and our datatype implementation with our ADT.
- The first component of our system will be our Lexer. We will be testing for the following types of tokens:
    - Header field elements: (C:, K:, L:, M:, Q:, T:, X:)
    - Strings in header ("Turkish March", "Mozart", etc.)
    - Types of bars ("|", "||", "|:", ":|")
    - Notes ("C", "D", "E", "c", "d", "e", etc.)
    - Rests ("z")
    - Octave descriptors (" ' ", " , ")
    - Note length descriptors (" 1 / 4", "2", etc.)
    - Accidentals ("^", "_", "=")
    - Chords ( "[ABC]")
    - Tuplets (" (3GAB", etc.)
    - Voices (" V: upper ", etc.)
    - Whitespace
    - Newline
    - Lyrics ("Amazing", "Grace", etc.)
    - Symbols to break up the lyrics ("-", "_", "*", "~", "\-", "|")
    - Invalid input (check for expected errors)
    - Three specific test cases:

        - "| A B/2 C/2 C^ [DE] |"
        - "w: Sa-ys my au-l' wan to your aul' wan, Will~ye come to the Wa-x-ies dar-gle?"
        - "X: 3
          T: Turkish March
          C: W. Mozart
          M: 2/4
          L: 1/8
          K: Am"
- The second component of our system will be our Parser. Our testing strategy is as follows:
    - Boundary cases (Measure with 0 notes, measure with 1 note, measure with many notes).
    - Cases testing the various descriptors of musical notes (Note with accidental, note with multiple accidentals, note transposed by one octave, note transposed by multiple octaves, note whose length is changed). Specific test case 2: "
    - Cases testing each musical component (Note, Rest, Voice, Chord, Tuple, Lyrics, Measure, Measure with a repeat, Multiple measures with repeats, multiple measures with a repeat, multiple measures with a repeat and alternate endings).
    - Cases testing lyrics (Fewer syllables than available words, more syllables than available words, a syllable aligned with more than one note, repeated lyrics for a repeated measure)
    - Cases testing ability to parse the header (Will test each of the 7 required header fields)
    - 3 specific test cases:
        - **"| z4 |",**
        - **"|: C/2 D/2 D 2E |[1 G^ [ACE] B c | G A B B :|[2 F E D C |"**
        - **"gf|e2dc B2A2|B2G2 E2D2|.G2.G2 GABc|d4 B2**
          **w: Sa-ys my au-l' wan to your aul' wan, Will~ye come to the Wa-x-ies dar-gle?"**
- The third component of our system is our ADT. Our test cases for the ADT are exactly the same as the test cases for the parser, with additional test cases testing for object equality. The ADT must be able to play a fragment of music from each one of these test cases.
    - As with the parser, we'll be testing for boundary cases, cases testing the various descriptors of musical notes, cases testing each musical component, and cases testing lyrics. We'll make sure that the ADT is able to play a fragment of music from each one of these test cases.
    - Additionally, we'll be testing for structural equality for the following classes: Chord, Duration, Measure, Note, ReadyToAddNote
    - 3 specific test cases:
        - **"|: C/2 D/2 D 2E |[1 G^ [ACE] B c | G A B B :|[2 F E D C |"**
        - **"gf|e2dc B2A2|B2G2 E2D2|.G2.G2 GABc|d4 B2**
          **w: Sa-ys my au-l' wan to your aul' wan, Will~ye come to the Wa-x-ies dar-gle?"**
        - Check if the representation of | A B/2 C/2 C^ [DE] | equals another copy of

a representation of | A B/2 C/2 C^ [DE] |.