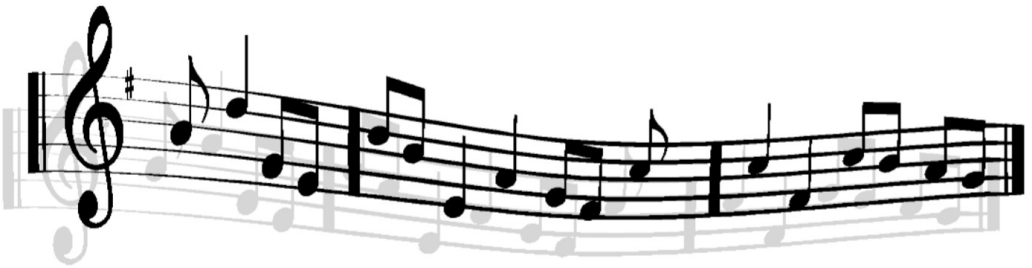
Project: ABC Music Player



Composition of a musical piece is often a trial-and-error process, in which the musician writes down a series of notes on paper and tests them out on a musical instrument. One way to do this on a computer is to type the notes into a text file using a special notation and feed them to a program that understands this notation. This way, you can transcribe your favorite pieces of music or compose your own pieces, and easily exchange them among your friends on the web.

abc is one of the languages designed for this purpose. It was originally intended for notating folk and traditional tunes of Western Europe, but it provides a sufficient set of constructs for transcribing a reasonably complex piece of music, such as a Beethoven symphony. Since its invention in 1980's, abc has become one of the most popular notations for music, with around 50,000 abc files circulating around the web.

In this project, you will build an *abc player*. You will use ANTLR to parse abc files, and the Java MIDI API to playback the music. You are required to handle only a subset of the abc language, which we will discuss in more detail in the specification below. This subset is sufficient to play a large number of interesting tunes that are available on the web, but you are welcome to implement the rest of the standard, as long as your overall design remains clean and simple.

In this project, you will practice working in a small group, using the software engineering techniques we have learned this semester: version control, specifications, unit tests, immutable data, abstract data types, and so on. You will also get more practice with grammars, parsers, and abstract syntax trees.

Design Freedom

On this project, you have complete design freedom to choose the packages, interfaces, classes, and method signatures you use in your code. Choose them wisely. You will be expected to use specs, tests, abstraction functions, rep invariants, safety arguments, `checkRep` and other assertions.

The specification in this handout constrains what your solution must do, but you will have many design questions that are not answered in this handout. You are free to come up with your own answers to these questions – just be reasonable, consistent, safe from bugs, easy to understand, and ready for change. Examples of questions that you won't find answers to here:

- Do we need to check the input for errors? What should we do if there are errors?
- Can tuplets or chords contain notes of different lengths?
- Should we handle nested repeats?
- What if two voices contain a different number of total bars?

…and there are many, many more.

Getting Started

To get started, download the assignment code and initialize a repository.

Overview

During the project, class meetings are replaced by team work time. Your team is expected to meet and work together during these times.

Checking in

Your team will be assigned a TA mentor who will help you with your design and help you stay on track as you implement it. You are required to check in with your TA during every team work time class. At this check-in, each team member should describe:

- what they have accomplished since the last check-in
- what they plan to accomplish by the next check-in
- what, if anything, is blocking their progress

Use the check-in to review how each person's plans fit together and decide how to resolve blocking problems.

Working together

Other than reflections at the end of the project, all parts of the project should be committed to the repository you share. Each commit to the repository should have a useful commit message that describes what you changed.

Use the code review skills you've practiced to review one another's code during the project.

You are also strongly encouraged to try pair programming, where two people collaborate on a single computer. Pair programming is a skill that requires practice. Be patient: expect that pairing will mean you write code *more slowly*, but the results are *more correct, more clear, and more changeable*. You can find plenty of advice on the Internet for how to structure your pairing.

When multiple people contribute to a commit, mention them in the commit message. Your TA will be reviewing the Git log to see individual contributions.

Pull before you start working, commit and push frequently, and don't break the build!

Dividing the work

Every team member must not only make a substantial contribution to the project, but *every* team member must make a concrete contribution to *every* major component of the project, as outlined in the project specification (leftmost column).

For example, you may not assign one team member who is solely responsible for the parser.

Contributions include writing specifications, writing testing strategy, writing tests, prototyping, writing internal docs, writing implementation code, fixing bugs, and giving code review feedback. Contributions you work on as a pair or a whole team are great, as long as everyone is involved. Tasks like running meetings, taking notes, and tracking bugs are important, but don't count as contributions under this requirement.

You must also divide the work such that *every* team member makes several *different kinds* of contributions.

For example, you may not assign one team member who is solely responsible for writing tests.

Tasks

- **Team contract.** Before you begin, write and agree to a **team contract**.
- **Understand the problem.** Read the **project specification** carefully.
- **Design.** You will need to write a grammar for parsing abc files; design an immutable abstract datatype (an AST) for music; design a module to convert parse trees into ASTs; design a module that sends music to the MIDI player; and other components.

  Your software design is perhaps the most important part of the project: a good design will make it simpler to implement and debug your system. Remember to write clear specifications for classes and methods; write data type definitions for your music data types; define abstraction functions and rep invariants, and write `toString` and `checkRep`; and document safety from rep exposure.
- **Test and implement.** You should write JUnit tests for the individual components of your system. Your test cases should be developed in a principled way, partitioning the spaces of inputs and outputs, and your testing strategy should be documented as we've been doing all along.

- **Reflection.** Individually, you will write a brief commentary saying what you learned from this project experience, answering the **reflection questions**. Your reflection may not exceed 300 words.

Specification

The **project specification** describes how the abc player must work.

The specification is not meant to provide you with comprehensive information on abc notation, so you may find the following links helpful as you work to understand the notation:

- abc standard v2.1 The current official standard for abc.
- Wikipedia article on abc.
- John Chambers' abc site: Another comprehensive site on abc. A great feature on this site is the abc tune finder, which lets you search through thousands of abc files around the web.
- Wikipedia article on modern musical symbols: A fairly comprehensive overview of the Western musical notation.

Provided code

`Abc.g4`, `Xyz.g4`	skeletons for the grammars you might use for parsing abc notation. You might use more or fewer grammars, and you should definitely pick better names.
`Configuration.g4`	common boilerplate that all your grammars can import and share
`SequencePlayer.java`	shows how to play music on your MIDI synthesizer
`Pitch.java`	data type representing a musical pitch
`Main.java`	placeholder for your main program

We also provide some example abc files that you can use to test your abc player. These files are found in `sample_abc/` in your Git repo.

`scale.abc`	A simple scale
`little_night_music.abc`	A Little Night Music by W. A. Mozart
`paddy.abc`	Paddy O'Rafferty, an Irish tune
`invention.abc`	Invention by J. S. Bach
`prelude.abc`	Prelude by J. S. Bach
`fur_elise.abc`	Fur Elise by L. v. Beethoven
`abc_song.abc`	Alphabet Song
`waxies_dargle.abc`	Waxie's Dargle

You can find many more abc examples online.

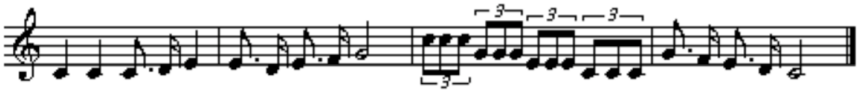
Warmup

Note: If you don't understand musical notation, this Wikipedia page may be helpful.

1. Transcribe each of the following small pieces of music into an abc file. Name your files `piece1.abc` and `piece2.abc`, respectively, and commit them under the directory `sample_abc` in your team's Git repository. You may find the spec useful

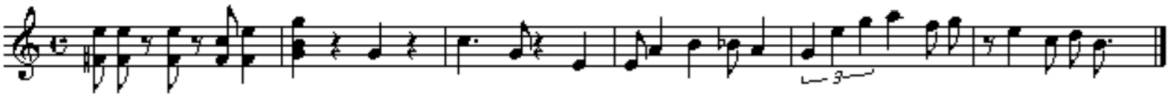
here.

`piece1.abc` : A simple, 4/4 meter piece with triplets.
As a starter, the header and the first two bars are already provided.
You should complete the rest of the piece by transcribing the last two bars.



```
X: 1
T:Piece No.1
M:4/4
L:1/4
Q:1/4=140
K:C
C C C3/4 D/4 E | E3/4 D/4 E3/4 F/4 G2 |
```

`piece2.abc` : A more complex piece, with chords, accidentals, and rests.
The piece below is only an excerpt, and the last measure is not a full measure.
Pad the piece with enough rests to complete the measure.
Use a default note length of 1/4, and make its tempo to 200 default note lengths per minute.



2. Write JUnit tests that play these pieces using the sequencer, similar to the main method in the `SequencePlayer` class. Put them in `test/abc/sound/SequencePlayerTest`.

   Hint: `SequencePlayer` has a `toString` method that produces a string representation of all its events. If you are not confident in your listening skills, this might be useful if you want to compare sequences that sound the same.

3. Write a data type definition and specifications of classes and methods for your immutable music data type. Commit them to your repository as skeleton Java class files.

Deliverables and grading

Day 1 Your team contract must be committed to your group repository in a PDF file called `team-contract.pdf` in the top level of your repo.

Day 5 **1st milestone meeting:** you will have a meeting with your group's TA mentor, at which you must demonstrate the three items described under warmup: (1) two abc files representing the sample pieces, (2) JUnit tests that play the sample pieces, and (3) a data

	type definition and specs for your music data type. **Commit code and specifications** to your repository. You will demo the sample pieces and discuss your work at the meeting. Have a laptop with the test program ready to run.
Day 10	You should have committed (1) a grammar for abc notation, (2) specs for all your classes and methods, and (3) some progress on testing and implementation. **2nd milestone meeting:** you will meet with your mentor on day 11 or 12.
Day 17	**Project deadline.** Your specifications, tests, and implementation should be complete and committed to your group's repository.
Day 17	**Reflection deadline.** Individually, you should write a brief reflection. Your reflection should be at most 300 words of plain text.

Grades will be determined according to the following breakdown:

- Team contract: 5%
- Design: 25%
- Implementation: 40%
- Testing: 25%
- Reflection: 5%

The Day 5 and Day 10 specifications deadlines are graded as binary checkoffs. Missing each of these intermediate deadlines will cost 5 points on the overall project grade. The associated milestone meetings also contribute to your design and implementation grades.

Check-ins with your TA mentor are also graded as binary checkoffs, either passed or missed. Missing a check-in costs 1 point on the overall project grade.

Hints

Pay attention to the various kinds of whitespace handling described in the grammar. Windows and Mac/Unix have different line endings (`\r\n` vs. `\n`), but the abc grammar requires you to support both kinds interchangeably. Do **not** use `System.lineSeparator()`, because it would make your code handle only one kind of line ending. Make sure you have appropriate test cases to avoid any platform dependency. Also make sure you are tolerant of extra spaces and tabs where appropriate.

You can break up the abc grammar into multiple ANTLR grammars. It may be easier to parse the overall structure of the file (the header and body section) with one grammar, then parse the details of the music itself with a separate grammar. Otherwise the terminals may overlap and confuse ANTLR.

It may also help to handle comments and spaces with ANTLR's `skip` command, to ignore them entirely. An example of `skip` can be found in the ps3 grammar, which uses it to ignore spaces.

Additional pages referenced above

ABC player specification

Team contract

Reflection