6.005 Elements of Software Construction
Fall 2011
Project 1: An ABC Music Player
Friday, October 14, 2011

Problem

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, such as a piano. 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* that plays an abc file, by parsing it and feeding it to the Java Midi API. You are required to handle only a subset of the language, which we will discuss in more detail below in the Specification section. 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.

Purpose

The purpose of this project is to help you gain experience in (1) designing and implementing programs in the functional style (that is, making extensive use of functions over immutable types); (2) designing and implementing abstract types; and (3) using common design patterns for constructing and traversing structures (such as Variant as Class, Interpreter, Iterator, and Visitor). It also introduces you to the fundamentals of compilation, in particular: (4) expressing a language as a grammar; (5) converting a text to an abstract syntax tree; and (6) organizing a compiler into phases (lexing, parsing, static semantic analysis, etc). Finally, the project will give you further practice in software engineering fundamentals, such as (7) clarifying a problem statement; (8) inventing clear and simple interfaces to minimize coupling, and identifying and resolving undesirable dependences; (9) structuring a program to make it easily testable, organizing, executing and evaluating test suites; and (10) working collaboratively in a team.

References

Before reading the problem specification, you should keep in mind that this document is **NOT** meant to provide you with comprehensive information on the abc notation. Instead, you should consult the following list of sites during your project:

- abc standard v1.6: Current official standard for abc.
- Required abc subset: a description of the required abc subset, including the grammar in an EBNF. For your interest, the full EBNF can be found here, but remember, you are required to implement **ONLY** our subset.
- Chris Walshaw's abc site: An informative web site by the inventor of the language. Among other things, the site contains a set of examples and a tutorial, which should help get you up to speed with 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 abc.
- Wikipedia article on modern musical symbols: A fairly comprehensive overview of the Western musical notation.

Specification

Note: You are **NOT** allowed to use any code taken from an existing abc player as a part of your implementation in this project.

Required Subset of abc. The subset of abc that you are required to implement in this project is described in a separate document, the abc subset for 6.005.

Tasks

You should perform the following tasks:

- **Team Contracts**. Please see this page for details.
- **Warmup**. There are some warmup exercises that you need to do, listed below. They will help you learn about how the abc music player works.
- **Grammars and datatypes**. Write out the grammar of the abc music player. Then specify which datatype definitions you would like to use for the different parts of the project.
- **Snapshot Diagram**. Write out diagrams of 3 distinct example ABC expressions. It should show what

happens in each part of the code in the middle of running the program.

- **Implementation**. Based on how your snapshot diagram, implement your code in Java. You may find that you want to make changes to your design, and thus your snapshot diagram. You are free to do this, but should record the changes so it is clear how and why you diverged from the original. Remember to do test first programming. You will need to write unit tests.
- **Test**. Test your entire system on the staff sample inputs. Create at least **three** additional test cases (e. g. your own abc files) to demonstrate that your player is able to correctly parse and play various musical constructs, and also detect any semantic errors in an abc file.
- **Reflection**. Write a brief commentary saying what you learned from this experience. What was easy? What was hard? What was unexpected? Briefly evaluate your solution, pointing out its key merits and deficiencies. This is an individual activity.

Infrastructure

Java MIDI Sequencer

Before you start the project, you should learn about the Java MIDI Sequencer. The Sequencer allows you to schedule a series of notes to be played at certain time intervals.

Look at the package `sound` under `src`, and study the provided class, `SequencePlayer`. For this project, you will not need to modify this class, but you should become comfortable using its constructor and the two methods, `addNote` and `play`.

addNote(int note, int startTick, int numTicks): This method schedules a note to be played at `startTick` for the duration of `numTicks`. Here, a "tick" is similar to a time step. At the beginning of a musical piece, the global tick is initialized to 0, and as the music progresses through the notes, the global tick is incremented by some number.

The first parameter `note` is a MIDI note value that corresponds to the pitch of a note. The provided class `Pitch` contains a number of useful methods for working with pitches. The method `toMidiNote` returns the MIDI note value of the particular note, and `transpose` can be used to transpose the note some number of semitones up or down.

SequencePlayer(int beatsPerMinute, int ticksPerQuarterNote): The constructor for `SequencePlayer` takes two parameters:

- The first, `beatsPerMinute`, is the tempo of a musical piece. It is expressed in the number of beats per minute (BPM), where each beat is equal to the duration of **one quarter note**. The BPM to be used for a particular piece depends on the value of the optional tempo field ('Q') in the input abc file. When this field is absent, the default tempo is 100 BPM, where each beat is equal in duration to the **default note length** (indicated by the field 'L').

- The second parameter, `ticksPerQuarterNote`, corresponds to the number of ticks per quarter note. Note that ticks used by the sequencer are based on discrete time. Think about how large this number needs to be in order to play notes of different durations in an abc piece. For example, if `ticksPerQuarterNote` had a value of 2, then an eighth note would be played for the duration of one tick, but you would not be able to schedule a sixteenth note for the correct duration.

play(): After all of the notes have been scheduled, you can invoke `play` to tell the sequencer to begin playing the music.

Run the main method in `SequencePlayer`, which shows an example of using a sequencer to play up and down a C major scale. In this example, all of the notes in the scale have been hard-coded. In your abc player, you will be walking over your data structures that represent a musical piece and automatically schedule the notes at appropriate ticks.

We are also providing some example abc files that you can use to test your abc player (included in the SVN directory `sample_abc`):

- `A simple scale (scale.abc)`
- `A Little Night Music by W. A. Mozart (little_night_music.abc)`
- `Paddy O'Rafferty, an Irish tune (paddy.abc)`
- `Invention by J. S. Bach (invention.abc)`
- `Prelude by J. S. Bach (prelude.abc)`
- `Fur Elise by L. v. Beethoven (fur_elise.abc)`

You can find many more examples online, including here.

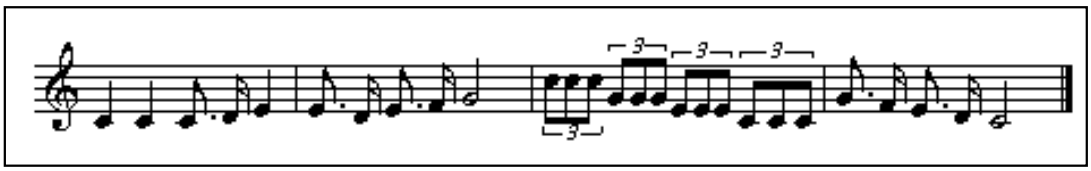
Warmup

abc Music Notation

Task 1: Transcribe each of the following small pieces of music into an abc file. Name your files as **piece1.abc** and **piece2.abc**, respectively, and commit them under the directory `sample_abc` in your team's SVN repository.

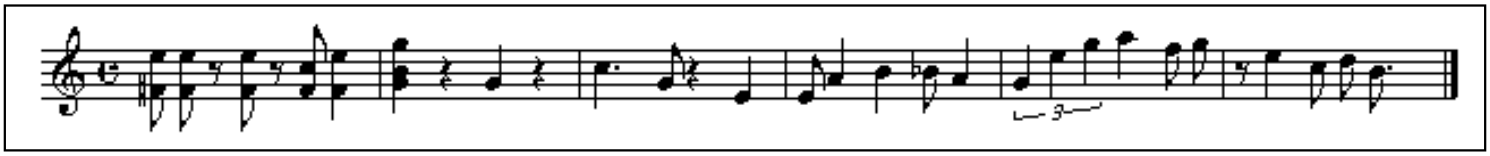
You may find the abc subset description useful.

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

Piece No.2: A more complex piece, with chords, accidentals, and rests. **Set its tempo to 200, with the default note length of 1/4.**



Task 2: Write JUnit tests that play these pieces using the sequencer, similar to the main method in the `SequencePlayer` class. Store them in a separate class called **`SequencePlayerTest`**.

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

Deliverables and Grading

For the first deadline (**11:59 PM, October 18, 2011**), your deliverables are:

- Team contracts, which should be in a file called TeamContract.pdf;
- Your results to the tasks in Warmup;
- Grammars, datatypes, state machines, and snapshot diagrams of the three given example ABC expressions, in one file called Design.pdf.

The deliverables for the main deadline (**11:59 PM, October 27, 2011**) are:

- the revised design in Design.pdf;
- the implementation;
- the tests;

All your code should be completed by this deadline!

The reflections due on the reflection deadline (**11:59 PM, October 29, 2011**) are:

- Reflections on Team. How did you feel the group did? How did your team work? How was the coding? How did you split the work?
- Reflections on Individual. How do you think you did? What did you do in the project? How do you feel about it?

Reflections should be committed to abc-reflections, which you will pull as if it's a pset.

Your code should be committed in the repository you share with your teammates by the deadline. All other parts of the project should be stored in your repository as two separate PDF documents, one for each deadline, as mentioned above. Each commit to the repository should have a commit comment saying what you changed, as well as who worked on it. Your TA/Mentor will be receiving your commit emails.

Grades will be allotted according to the following breakdown:

- Team Contract -- 5%;
- Design -- 25%;
- Implementation -- 50%;
- Testing -- 15%;
- Reflection -- 5%

Hints

Start early! This project is more work than it seems. Starting early on the project will give you more time to sort out any issues and ask the staff questions that may arise, especially if you have trouble with transcribing music.

Lexing and parsing: Given the grammar for abc, you will need to build a lexer that breaks the input string into tokens and a parser that groups the tokens into a valid syntactic construct and produces an abstract syntax tree (AST).

Designing datatypes: For the design of your abc player, you should think **carefully** about datatypes that you need to represent the musical constructs in abc. Start with simple constructs, such as notes and rests, and think about how you would build up on these primitive objects to create more complex structures. How would you represent a triplet? A chord? What does a bar consist of? How would you represent multiple voices? Sort out answers to these questions with your team members during the design stage.

Evaluating expressions: Once you parse an abc file and create your own internal representation of the music as an abstract syntax tree (AST), you will need to perform various computations that involve traversing the tree, possibly multiple times. Consider carefully the patterns you learned for performing these traversals: you may want to use the visitor pattern, though an interpreter pattern might also work.

Parsing and pattern matching: We recommend that you think carefully about your approach to parsing the abc file; consider how to make it easiest to write a single parser that can handle the entire file, with complexities such as bars split across lines. You may find the Java pattern matching libraries such as `java.util.regex.Pattern` and `java.util.regex.Matcher` helpful, but you should use them judiciously.

Multiple voices: A particular challenge you should think about is how you will represent multiple voices, and how you will merge them into a single sequence of midi events.

Use classtime wisely: Lecture and Recitation are cancelled for team/TA meetings for the duration of the project. You should use the time to work on the project, or meet up with your TA/mentor. This mentor is randomly assigned, so may not be the one who teaches your recitation.

6.005 Elements of Software Construction

Fall 2011