6.005 Elements of Software Construction
Fall 2008
Project 2: An ABC Music Player

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 (5) 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 (6) clarifying a problem statement; (7) inventing clear and simple interfaces to minimize coupling, and identifying and resolving undesirable dependences; (8) structuring a program to make it easily testable, organizing, executing and evaluating test suites; and (9) 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, although you may study existing implementations and incorporate features of their design if you want to.

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.

Semantic Analysis. In addition to making sure that an input file is *syntactically* well formed with respect to the provided EBNF, you must also carry out checks to ensure that the music is *semantically* consistent. This phase is usually called *static semantic analysis* in a compiler. For example, one type of analysis that you might want to do is to check that the sum of the durations of notes in every bar is equal to the value that is represented by the meter (although this particular rule has some exceptions). In your specification, you should define a set of semantic checks and how you plan to deal with them, by displaying intelligible error or warning messages when problems are encountered, and either terminating without playing the piece or playing it as best you can.

Interface Requirement. Your abc player should be a stand-alone Java application that takes as its only input argument the name of the abc file to be played. In addition to playing the music using the Java Midi API, it should also write the header information to the standard output stream so it can be displayed on a console.

Tasks

You should perform the following tasks:

- **Laboratory**. Complete the lab for this project. A separate handout describes the lab activities.
- **Abstract Specification / Design**. Describe the abc input files and the structures you plan to build from them using grammars, state machines and datatype productions. Note that the standard grammar will likely not be in a suitable form for parsing, and you may need to formalize aspects of the input file that are usually described informally (such as the lexical structure). List the semantic checks you will perform, and describe how violations will be treated.
- **Code Design**. Sketch the code structure using dependency diagrams for **two** possible

implementations of your abstract design that differ substantively. List their relative merits, select one of them, and explain why you chose it.

- **Implementation**. Implement your chosen code design in Java. You may find that you want to make changes to your design. You are free to do this, but should record the changes so it is clear how and why you diverged from the original design.
- **Test**. Run the staff test suite using the test framework provided. 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.

Infrastructure

We are providing you with the `SequencePlayer` class, which you may use to play a sequence of notes in your abc player. The `SequencePlayer` is essentially a scheduler. Its `addNote(int note, int tick, int duration)` method allows you to schedule a note (whose pitch is represented by `note`) starting at some time step (`tick`) for a certain `duration` (the number of ticks for which this note should be played). You may also use the provided `Pitch` class to obtain a numeric value of the pitch for a certain note (look at `toMidiFrequency()` method).

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.

Deliverables and Grading

For the first deadline, your deliverables are:

- the abstract specification/design;
- and the code design;

Your TA will arrange to meet with your group during lab time to discuss your design deliverables.

The deliverables for the second deadline are:

- the implementation;
- the tests;
- and your reflections.

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.

Grades will be allotted according to the following breakdown: abstract design -- 25%; code design -- 25%; implementation -- 30%; testing -- 10%; reflection -- 10%.

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.

Lexing and parsing: In some ways, this project is similar to the implementation of the multi-unit calculator from Problem Set 1. 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.

Designing datatypes: The evaluation of an expression in the multi-unit calculator from Problem Set 1 was relatively simple. However, for the design of your abc player, you should think much more **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 in lecture for performing these traversals.

Parsing and pattern matching: We recommend that you follow the JSP approach to parsing the abc file; this will make it easiest to write a single parser that can handle the entire file, with complexities such as bars split across lines. You may also 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.