McMaster University Comp Sci 4TB3/6TB3, Winter Term 2017/18 — Lab 5
For the Labs on Febuary 6 - 9,
Due Monday, February 12, 11 pm

Eden Burton, Spencer Park
out of 24 points; extra points count towards any other lab

- *Submission is to be done exclusively through Avenue. Submissions via e-mail will not be accepted. A* ***10% penalty*** *will be accessed for each day the lab is submitted after the due date.*
- This lab will require `antlr 4` and `java 6+`. The latest version of antlr (v4.6) can be downloaded from `http://www.antlr.org/`.
- In this lab, you are allowed to work in pairs, provided that you split the work equally and arrive at a common understanding of the solution. However, in that case you must state in your submission the person you worked with, such that similarities in the solution will not be construed as Academic Dishonesty. Working in groups of three or larger is not allowed and will be considered Academic Dishonesty. If you look for someone to work with, we will try to find a match, please contact the TAs.
- You are allowed and encouraged to talk to everyone in the course to get a common understanding of the problem, but you can share partial solutions only with your collaborator, if you work in a pair. The final submission must be your own, that is, you cannot submit identical submissions with two names on them.
- The Tutorial Exercises will be presented in the tutorials; you need to submit only answers to the Lab Questions. In the labs, the solution to last week's lab questions are discussed and you can get help with this week's lab questions. Attendance at the labs is not checked.

Tutorial Exercise 1 (Python Object-like Macros)**.** An Object-like macro is an identifier that will be replaced by the value that follows when the identifier appears elsewhere in the program. We want to add this feature to python using antlr to write the preprocessor. The input and output for the program will be a python script with the output stripped of `#define` and the references replaced.

See `example/PythonMacros.g4` on avenue for the antlr grammar.

You should have the `antlr4` and `grun` commands set up on your system which can be found on the `http://www.antlr.org` page.

Running the example can be done via the 2 commands:

```
antlr4 PythonMacros.g4 -no-listener
javac *.java
# grun <grammar> <startRule> [input-file]
grun PythonMacros program sample.py > out.py
```

Answer.

Lab Question 1 (MIDee Implementation, 24 points)**.** MIDI is a protocol that allows computers and electronic musical instruments, such as a synthesizer, to speak with one another. Each MIDI message describes an instruction for the synthesizer to execute. A nice summary of messages can be found at `https://www.midi.org/specifications/item/table-1-summary-of-midi-message`.

This lab will involve generating audio from a program written in the language defined below. We will generate sounds based on 2 parameters: pitch and duration. **Pitch** changes what note we hear and **duration** describes how long we hear it for. In MIDI there are 128 notes ranging from `c0` to `g10`. Picture each note as a switch that can be turned on or off. A note can also be made **sharp** or **flat** by raising or lowering the pitch 1 semi-tone (1 note number) respectively. Here is a link to a table that displays the mapping between a note name and octave to it's number `http://www.electronics.dit.ie/staff/tscarff/Music_technology/midi/midi_note_numbers_for_octaves.htm`.

notation	pronunciation	MIDI number
`c0`	`"c zero"`	0
`g#3`	`"g sharp three"`	44
`e_7`	`"e flat seven"`	87

Figure 1: Note to number examples

Our language (MIDee) is made up of 2 statements:

Play statement: plays 1 or more notes for a given duration in beats. The statement looks like the keyword `"play"` followed by 1 or more *notes* separated by a comma `","` followed by the keyword `"for"` followed by a *duration* in beats terminated by a semi colon `";"`

Wait statement: waits for a given duration in beats. The statement looks like the keywords `"wait"` `"for"` followed by a duration in beats terminated by a semi colon `";"`

Notes are a single lower case letter `"a"` through `"g"` optionally followed by a sharp or flat (`"#"` or `"_"`) terminated by a required octave (0 through 10). Some examples can be seen in the *Note to number examples* table. You will need to convert from name to MIDI number. If the note is out of bounds (below 0 or above 127) a warning should be printed and the note rounded to the nearest legal value.

A *duration* is a number describing time in beats (See the `getDurationInTicks` method). It can be written as a whole number, floating point number.

The `play` and `wait` statements may only appear inside an instrument scope. To play some statements on an instrument the code looks like an *instrument name* optionally followed by `"@"` *tempo* with an opening and closing brace, `"{" "}"`. If the instrument or tempo is different than previously set the compiler should call the `setInstrument` or `setTempo` methods to make the change. For example, if 2 consecutive instrument scopes specify the same tempo of 120 only the first set the tempo as the second would be a waste and just filling up the output with an extra message.

An *instrument name* is 1 or more letters. These do not overlap with the keywords or single letter strings of `"a"` to `"g"`. i.e. `"for"` and `"b"` are not instrument names.

The *tempo* describes the speed of the playback in beats per minute. The default is 120bpm. In our language a tempo is a positive whole number.

White space is not meaningful in this language. See avenue for a sample MIDee program.

More formally the grammar for the language with repetition as '∗', optional as '?', and **whitespace allowed to separate all tokens** (see the `skip` command):

⟨program⟩ ::= ⟨instrumentBlock⟩* ⟨EOF⟩

⟨scopeHeader⟩ ::= ⟨INSTRUMENT⟩ ('@' ⟨NUMBER⟩)?

⟨instrumentBlock⟩ ::= ⟨scopeHeader⟩ '{' (⟨playStatement⟩ | ⟨waitStatement⟩)* '}'

⟨playStatement⟩ ::= 'play' ⟨note⟩ (',' ⟨note⟩)* 'for' ⟨duration⟩ ';'

⟨waitStatement⟩ ::= 'wait' 'for' ⟨duration⟩ ';'

⟨duration⟩ ::= ⟨NUMBER⟩ | ⟨FLOATING_NUMBER⟩

⟨note⟩ ::= ⟨NOTENAME⟩ ('#' | '_')? ⟨NUMBER⟩

Task: Write an antlr grammar (using the template `MIDee.g4`) that parses a MIDee program and compiles it into a MIDI file on `System.out`. Your grammar must use the helper methods defined in `MIDIHelper.java` for manipulating the output sequence.
Submit: A single `MIDee.g4` file on Avenue.

Your solution should be able to be built with the following commands:

```
# compile the grammar and put the generated source into the parser
   package
java -cp antlr-4.6-complete.jar org.antlr.v4.Tool -o src/cas/cs4tb3/
   parser -no-listener MIDee.g4

# create a directory for the output
mkdir build

# compile the source which includes the generated files from antlr
javac -cp "antlr-4.6-complete.jar" -d build -sourcepath src src/cas/
   cs4tb3/MIDee.java

# bundle the compiled classes into a jar
jar cfm MIDee.jar src/META-INF/MANIFEST.MF -C build .
```

when sitting in an environment with the following structure (these files can be found on avenue

```
src/
  cas/
    cs4tb3/
      GeneralMidiInstrument.java
      MIDee.java
      MIDIHelper.java
  META-INF/
    MANIFEST.MF
antlr-4.6-complete.jar
MIDee.g4
```

The solution can then be run with

```
    java -jar MIDee.jar < sample.midee > out.mid
```

and the output file played in your regular media player (if it supports MIDI), using a tool such as `timidity` to convert it, or using an online app/extension etc.