Homework 4

CMPT 341 — Fall 2018

This homework is a little different from most. In this one you'll be building a small scanner-parser engine for a (very) simple language. **You are to work in groups of 2–3 on this homework. Only one student needs to submit, but make sure everyone's name makes it onto the submission.**

There are two parts of this homework: written and code. Please turn in a single PDF for the written part and a zip file for the code. The code files to include are (depending on how much you completed):

- #3: your .jflex file containing the scanner specification;
- #3: the .java scanner class generated from the .jflex;
- #4: your .jcup file containing the grammar specification;
- #4: the .java parser class generated from the .jcup;
- #4/#5: a .java program that reads a file and reports that it is valid or not according to the parser (and, if you completed #5, that builds an AST for the input);
- #6: a .java program that acts as an interpreter for the language (if you completed the extra credit).

Language description

You are to write a scanner and parser for a simple language that performs calculations. Example pieces of code in this language to give you a feel for it:

- ```
  print 15-2*3;
  ```

  (prints `9.0`)

- ```
  g = 9.81;
  t = 5;
  height = 1000 - 1/2*g*t^2;
  print "The height after ", t, " seconds is ", height, " meters.";
  ```

  (prints `The height after 5 seconds is 877.375 meters.`)

- ```
  a = input "Enter one leg: ";
  b = input "Enter the other leg: ";
  c = sqrt(a^2 + b^2);

  print "The length of the hypotenuse is ", c, ".";
  ```

  Running, the program would look like this (user input in blue):
  ```

```
Enter one leg: 7.2
Enter the other leg: 11.9
The length of the hypotenuse is 13.9086.
```

In this language, variables don't need to be declared (although they do need to be assigned a value before being used). Variables names are similar to Java — one or more letters, digits, or underscores, where the first character is not a digit. All variables are automatically numeric (double) type, so you can't assign a string to a variable. Variable assignments are of the form "varName = value".

Arithmetic is as usual (+, -, *, /, ^, ()) with the standard precedence and associativity rules. There are also sqrt, sin, cos, and tan functions that each have a single argument.

There are two special functions: input and print. Input takes a string parameter and returns a numeric input value; print takes one or more values (numbers, variables, or strings) separated by commas and prints them out.

A statement in this language is terminated by a semicolon. There are two different kinds of statements: assignments (varName = value) and print statements. Whitespace is ignored as in Java.

Note on running Java Cup

When you extract Java Cup you'll find that it consists of two .jar files. (I suggest that you place these directly in your homework folder, alongside your .jflex and .jcup files.) The first, `java-cup-11b.jar`, is an executable jar file containing the program itself. To run it, do

```
java -jar java-cup-11b.jar name-of-.jcup-file
```

from a command line.

You will also need to include `java-cup-11b-runtime.jar` in your classpath when you compile/run code using the parser. To do this, use the Java -cp flag:

```
javac -cp .:java-cup-11b-runtime.jar *.java        # Compile in Linux / OS
javac -cp .;java-cup-11b-runtime.jar *.java        # Compile in Windows
java -cp .:java-cup-11b-runtime.jar name-of-class  # Run in Linux / OS X
java -cp .;java-cup-11b-runtime.jar name-of-class  # Run in Windows
```

Problems

Written part

1. (30%) Write down lexing rules for this language. That is, identify every possible kind of token in the language and give a regular expression description of each. (There are plenty of online tools to help you with regular expressions.)
2. (30%) Write down a grammar in BNF or EBNF form for this language. Use the token types

from #1 as the terminals in your language.

Coding part

3. (20%) Use JFlex to make a scanner (lexer) for the language. I suggest you use the example from class as a starting point.

   (Note that the .jflex file uses symbols and classes generated by Java CUP – in algebra.jflex, the AlgebraParserSym class, for example – so although you should be able to run JFlex before Java CUP, you won't be able to compile the generated .java file until after completing the next step.)

   Your scanner will need to handle strings correctly. The JFlex documentation has a simple example that illustrates how to do this (under "A simple Example: How to work with JFlex") – notice the `%state STRING` line, then subsequent `<STRING> { ... }` stuff. This code also implements escape sequences (like `\n`) which you are free to use or not.

4. (20%) Use Java CUP to generate a recognizer for the language. (If you don't know how to extract a .tar.gz file, here's a zip version of Java CUP.) Again it's probably a good idea to use the class example as a starting point.

   At this point you're just making a recognizer – a program that determines whether input is valid in the language or not. What this means for our purposes is that you can just delete all the `{: RESULT = ....; :}` stuff from the rules in algebra.jcup. It will still parse the input (and display an error message if there's a syntax error) but it won't build an AST.

   Keep in mind that Java CUP does not support EBNF syntax, just basic BNF. If you used any EBNF features ({ } or [ ]) in #2, you'll need to rephrase those into their BNF counterparts here.

   To get full credit for this part, in addition to your .jcup file you will also need to include a driver program (such as the class example AlgebraParserTest.java) that inputs a file and prints out whether the file is valid syntax or not.

5. (+50% **extra credit**) Build a full parser for the language that not only recognizes, but also forms an AST. The details of how this works are up to you, but you might do something like the example from class. You will get full credit for this problem if your scanner and parser recognizes syntax correctly and the scanner also generates a complete AST of an input file.
6. (+50% **extra credit**) Use your parser from #5 to implement an interpreter for this language. Your program should parse an input file and then execute it – saving values for variables, doing inputs and prints, and so on.