Exercise9: ANTLR

1 Objectives

Using ANTLR, you will create a Java based lexer and **parser** that reads in an expression from the command line and outputs the value of that expression.

ANTLR reference can be found at http://www.antlr.org

The steps are:

1) Create/Edit a ".g4" file which consists of LEXER and PARSER rules.
2) Use ANTLR on the ".g4" file to automatically GENERATE java codes for the lexer and parser.
3) Use the Parser as you see fit. It will return a PARSE TREE. You can traverse it to analyze it or to generate CODE.

We have several EXAMPLES that you should first view and tryout and understand. That will give you a good idea of the format of the ".g4" files.

ANTLRWORKS is a tool (shown in class) that you can download from http://tunnelvisionlabs.com/products/demo/antlrworks

2 Warm Up: Try Some Examples

1. First, open blackboard, go to Course Contents, and then download exercise9.zip file into your workspace (U:\workspace or something like that!). Then, unzip.
2. Browse through the the pdf file on antlr that has been provided. It will help you understand the background.

2.1 Setting up Antlr (same as Lexer Lab)

- The antlr-4.5.3-complete.jar file is located in your unzipped folder.
- You will need to make sure you have a jdk (i.e. with you have a java compiler). The path to the jdk must be on your PATH environment variable.
- Both the current directory and also the antlr jar file must be on your CLASSPATH variable. Using System Properties dialog > Environment variables > Create or append to CLASSPATH variable with the path to the jar file you copied above.
- Naresh created antlr.bat & grun.bat (which are in the examples directory) so that you can start using the commands antlr and grun on windows.

2.2 Running the examples

Each folder has three files: a .in file , a .g4 file, and a TestGrammar.java . The .in is the input file, the .g4 has grammar rules, and the TestGrammar uses an API to be able to run the parser from Java code and give you access to the parse tree from your java code.

You will be running the following commands on each of the example grammar files.

a) antlr4 <<Grammarfile.g4>> // this will create java files for lexer AND parser
b) javac *.java // this will compile and create appropriate class files
c) grun <<GrammarName>> start **-gui** < <<input-file>>
start refers to the start variable of your grammar.
-gui tells the test rig to show a screen with parse tree.
d) Also, grun <<GrammarName>> start **-tree** < <<input-file>>
-tree tells the test rig to show the parse tree in text form

Here is an example of the steps you will need to take

```
Step 1 : antlr4 HelloWorld.g4  // this creates the java program
Step 2 : javac *.java
Step 3 : grun HelloWorld start -gui    < HelloWorld.in   // this runs the parser
Step 4 : grun HelloWorld start -tree   < HelloWorld.in  // also runs parser
```

Now, take a look at the TestGrammer.java program to see how the lexer and parser are invoked from a program to get access to the parse tree **programatically**!

Run TestGrammer by typing java TestGrammer HelloWorld.in HelloWorld.g4 start

Go to the NEXT exercise and try out all the steps (i.e. steps1-steps4 above).
Do this for ALL the exercises.

NOTE1: Try and fix syntax errors by fixing the input files

3 SIMPLE CALCULATOR USING ANTLR

3.1 Reverse Polish Notation or Postfix Notation.

http://en.wikipedia.org/wiki/Reverse_Polish_notation

NOTE: MAKE SURE TO RUN EXAMPLE ex7_Expr as that will show you how to get the values from the parse rules.

Write antlr grammer rules (i.e. create a ".g4" file) that accepts multiple expression STATEMENTS written in Reverse Polish Notation. Your grammer must

- support numerical operations i.e. +, -, *,/, and %, and
- also logical operations i.e. &&, ||, ! , and
- relational operations (<, <=, ==, !=, >, >=).
- Skip white space.
- Statements are to separated by ";".

You can assume your language will accept only INT and BOOLEAN types. Also, assume that all operations (except !) have two operands.

Your parser will also have to **EVALUATE** each statement. You have the ability to add code as you "find" each operator and each operand. One way to implement evaluation is to use a stack. Stack<int> temp_stack = new Stack<int>(); Push operands as you see them and when you see an operator, pop them and operate on them and push the result.

A RPN.g4 file has been provided which you can use to start your solution.

Example of input file:

10 20 + 30 * ; 10 20 <= 13 20 >= && true || ; 10 20 - ;

Your output:

Aside from checking for syntax errors, it should also print the results:

900;true; -10;

4 Submission:

Zip your files (the .g4 file for RPN and participation file (i.e. who worked on which part or if you worked together). Then, submit on black board. Remember there is only one submission per group. Make sure to include all the files that are needed in order to run your program.