Final Project Report

I. Introduction

The goal of assignment 1.1 was to create a program that analyzes and parses expressions in Reverse Polish Notation, often referred to as postfix notation. Instead of designing a framework, the assignment requires using a standard tool for syntax analysis and parsing.

II. Approach

My research into standard tools for syntax analysis and parsing led me to a free framework, ANTLR4 (ANother Tool for Language Recognition 4) [1]. This tool handled both analysis and parsing, and it would be more than sufficient in handling RPN expressions. ANTLR's website includes testimonials from professionals such as Brad Cox and Samuel Luckenbill, describing ANTLR as "clean and concise", "efficient and stable" and an improvement over tools such as YACC or Lex [2].

ANTLR is implemented with Java, but it also targets C#, C++, JavaScript, Python, Swift and Go. Because Java is required for ANTLR's grammar parsing, my solution uses Java in order to limit the amount of external tools required.

Notably, the advantages of ANTLR included a readable grammar definition file and a visitor software design pattern to seperate the parsing algorithm from the parser's class structure [3]. This would help make the code for my solution more organized and readable. The disadvantage of this class structure is the bulkiness and repetition. This is an acceptable cost for larger use cases, but for the simplicity of this assignment, it made my solution more complicated than necessary.

III. Implementation

The implementation can be described with three distinct parts: The grammar file, the parsing algorithm (BaseVisitor Class) and the entry class utlizing the framework.

The grammar file consists of rules for identification and parsing [4]. For parsing RPN, many of the rules are recursive. The goal is to recursively parse matched rules into the last atomic rule. Instead of matching characters, some of the rules use regex-style syntax to detect a token. For example, ID uses "[a-zA-Z_]" to match a letter or underscore. The hashtags after a pattern in the rule definitions are matched to methods in the visitor classe to determine how they are parsed. For example, the line with "# Addition" will be matched to the "visitAddition" method.

```
15  LPAR  : '(';
16  RPAR  : ')';
17
18  NL    : '\n' | ';';
19  WS    : [ \t\r]+ -> skip;
20  ID    : [a-zA-Z_][a-zA-Z_0-9]*;
```

(a) Grammar token definitions

```
25  input
26      : setVariable NL input # ToSetVariable
27      | addOrSub NL? EOF     # Calculate
28      ;
29
30  setVariable
31      : ID EQUAL addOrSub # Assignment
32      ;
33
34  addOrSub
35      : addOrSub (mulOrDiv)+ ADD # Addition
36      | addOrSub (mulOrDiv)+ SUB # Subtraction
37      | mulOrDiv              # ToMulOrDiv
38      ;
```

(b) Grammar rule definitions

One of the design issues encountered in the grammar rules was the option for operators like addition to have more than two terms. This choice makes the expressions more concise and convenient, but it forces the use of parenthesis for grouping. Operators with multiple terms are allowed in my solutions in order to accomodate sample input given in the assignment description which utilized them. This is accomplished by wrapping one of the terms of the operator with parenthesis and a '+' to indicate one or more can be matched.

```
23      @Override
24  @   public Double visitDouble(RPNCalculatorParser.DoubleContext ctx) {
25          return Double.parseDouble(ctx.DOUBLE().getText());
26      }
27
28      @Override
29  @   public Double visitAddition(RPNCalculatorParser.AdditionContext ctx) {
30          Double result = visit(ctx.addOrSub());
31          for (RPNCalculatorParser.MulOrDivContext c : ctx.mulOrDiv()) {
32              result += visit(c);
33          }
34          return result;
35      }
```

Figure 2: Visit method example

The base visitor class contained methods that handled each operation. ANTLR's visit method was used to recursively evaluate expressions until they are parsed to a number. Each rule is given a supporting context class, which can be used in the visit implementations to determine the tokens or groups of tokens specified in the rule.

The parse tree and visit algorithms are used on expressions in the main class. ANTLR lexes, tokenizes and creates a parse tree from the expression, and the base visitor class is used to recursively parse the tree into a single node. Notice how ANTLR can use the grammar to create a parse tree out of the expression without any foresight about the visit algorithms that will be used to parse through the tree.

```
// Create parse tree with expression
RPNCalculatorLexer lexer = new RPNCalculatorLexer(CharStreams.fromString(s));
CommonTokenStream tokens = new CommonTokenStream(lexer);
RPNCalculatorParser parser = new RPNCalculatorParser(tokens);
ParseTree tree = parser.input();

// Output results of parsing tree
RPNBaseVisitor baseVisitor = new RPNBaseVisitor();
Double result = baseVisitor.visit(tree);

System.out.println("Expression: " + s + "\nResult: " + result + "\n");
```

Figure 3: Usage of parse tree on expression

The final step was to package the project for use. The main issue here was that ANTLR generated source code after parsing the grammar file, and compilation required these files and ANTLR's jar package to be in the classpath. The most ethical option appeared to be to include my source files and ANTLR's free jar package, and have them built with a bash/batch script for Unix/Windows systems respectively. The build scripts execute the program with the included test data, but the program can be run with other input by following instructions in the included README markdown file. This choice was made to seperate my solution's code from dependency code, follow submission guidelines on not including compiled binaries and make the build/execution as simple as possible.

IV. Conclusion

ANTLR is correct in calling itself a powerful parser. ANTLR's object-oriented design made my solution quick to write and easy to scale up or down. The complexity of such a substantial framework is wholly unnecessary for a project on the scale of this assignment, but it is not an unreasonable level of complexity given the ease of writing the grammar and parsing algorithms. Developing with ANTLR is still viable, given the project's development is still active and there are a breadth of resources about it available for free. From the experience of this assignment, given a task of creating a parser for a programming language, ANTLR would be the first choice. For a simpler application of a language parser, more lightweight tools such as YACC or Lex would be more fitting.

V. References

1. https://en.wikipedia.org/wiki/ANTLR
2. https://www.antlr.org/
3. https://en.wikipedia.org/wiki/Visitor_pattern
4. https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md