Jay Offerdahl - 2760730
Lab Time: Friday, 4:00pm - 5:50pm

Lab 5 Report

The objective of this lab was to create a lexer and parser that takes in an expression, and outputs the value from that expression. The lexer and parser were to be Java based, and were created from the ANTLR parser generator tools. Similar to our previous labs in which Lex was used, I implemented regular expressions fragments in order to create more complex lexical rules. These lexical rules were, in turn, used by the grammar I coded (which, in turn, was converted into a parser by ANTLR).

An LL(*) parser is what I've accomplished with this lab. Both my program and a normal LL(*) parser scan input from left to right, using a leftmost derivation scheme. These two characteristics make up the "LL" in the name. Next, the asterisk stands for a variable number which signifies the number of input symbols to look ahead at each step in order to make a parsing decision. This is to be contrasted with an LL(k) parser, however. With an LL(k) parser, the number of lookahead tokens it limited to k. With an LL(*) parser, there is no restriction on the number of lookahead tokens when parsing. The reason this is possible is because parsing decisions are made by recognizing if the tokens fit into the language specified by the grammar.

For a high level discussion, the input is taken in from the command line, tokenized based on our defined tokens and lexical rules, and then fed into the parser to see if it matches the grammar I have defined. If the input does match the grammar I have defined, a result will be outputted to the console.

In order to create a functioning calculator, mathematical order of operations must be followed. The design that I'm presenting along with this submission correctly breaks down the order of operations from the top being the least priority order, all the way to the bottom, where terms are translated to their actual double (float) values. You will notice that I kept track of each level of precedence of the order of operations by using the helpful names from "Please Excuse

My Dear Aunt Sally", in which Please = parentheses, Excuse = exponents, My = multiplication, Dear = division, Aunt = addition, and Sally = subtraction. Obviously addition and subtraction have the same precedence, and multiplication and division are on their own level of precedence.

Aside from keeping order of operations in check, I also had to make sure the correct nonterminals were linked to inside each rule. For example, precedence is followed up the chain, first from +/-, to * and /, to exponents (^), to parentheses. At parentheses, however, the chain starts over, unless the current expression is a logarithmic or trigonometric function (or a term, for that matter). This is to handle cases where simpler functions are contained in parentheses.

Finally, terms are converted to values based on the lexical rules I defined earlier in the program. Binary numbers start with 0b, so in order to get that value from the input, I have to trim two characters off the front of a binary string, and convert to double. The same goes for octal, except I only trim the leading 0 off. Both real/decimal values are an easy conversion, and hexadecimal values also require trimming two characters off the front of the string pre-conversion.

While writing and testing this program, I ran into several errors stemming from the correct order of operations (which was trivial to fix). My first mistake was pairing exponents and parentheses together in the same rule. This obviously caused problems because they lie at different levels of precedence. Next, I originally thought to put log, sin, cos, and tan at the beginning, because they may be operated on before functions. I later found that these have the highest level of precedence (besides terms) because their operands need to be operated on.

As far as useful ANTLRWorks features, I found the syntax diagrams especially useful when creating my rules. This allowed me to see the loops back on rules that could have several continuances of the same derivation (x + y - z + …). This was also useful to see the '()' being placed around the “auntSally” term (addition/subtraction) before continuing. I will admit, I was not the biggest fan of the text editor included, although, I didn’t spend too much time customizing it.