Lynne Coblammers
EECS 665
Lab 6 Report
2015.10.15

For this lab, a lexer and parser were created for a simple calculator language using ANTLR parser generator tools. The lexer and parser takes input from the command line and can handle binary, octal, hexadecimal, decimal, and floating point numbers. It can also handle arithmetic operations, exponents, natural log, sin, cos, and tan, and combinations of these including parenthesized expressions, evaluating expressions with the correct precedence.

ANTLR produces LL(*) parsers. These work by scanning the input from left to right and generating a left-most, or top-down, derivation. LL(*) parsers use either recursive descent parsing or a predictive parsing table. Recursive descent parsing is implemented by having a recursive procedure for each grammar rule that looks at one or more look-ahead symbols to determine what action to take. To make progress in parsing, the recursive procedure can't immediately call itself and must be able to determine what action to take based on the lookahead symbol or symbols. This means that grammar rules cannot contain left recursion and must be left factored.

These limitations were important in developing a grammar to recognize the calculator language. The first step was to write the lexer rules that would turn the input into a series of tokens. Rules were written to recognize each of the possible input symbols, which was pretty straight forward. For example, a binary number was recognized by looking for the string '0b' followed by any number of 1's and 0's.

Next, the grammar rules were written. The general design behind the grammar rules was to have rules producing statements with lower levels of precedence calling into rules with higher levels of precedence. This meant the rule for addition and subtraction (term 3) called into the rule for multiplication and division (term 2) and so forth. Unary operators including logarithm and the trigonometric functions were considered to have the highest operator precedence, but parentheses were the final level since they override all other precedence rules. Since any type of operation could be inside parentheses, this rule has parentheses surrounding the lowest level rule, term 3. There was also a rule for a number, which could resolve to any of the number types supported.

Once all of the basic rules were written, the program needed to be debugged and actions for each rule needed to be added. ANTLRWorks was very helpful in this process. First of all, the program highlighted syntax problems with the rules I had written, including order of definition and small syntax errors such as missing semicolons. I needed to reorder the rules so rules were defined before being referenced elsewhere. I also had initially written the rules using left recursion, which had to be eliminated. ANTLRWorks had an option to automatically eliminate left recursion, which was very helpful. Another very useful feature was being able to see the parse trees in action using the debugging feature, which helped me recognize the need for parenthesized rules to allow any other type of rule to be inside the parentheses. ANTLRWorks also provides visualization of the grammar rules, which helped me eliminate ambiguities in my rules by allowing me to more easily see how the ambiguities arose.

Once the parse trees looked correct, actions were added to each rule to calculate the value of the rule. This was done using Java to interpret the number strings appropriately (e.g. base 2 for binary numbers), turn them into doubles if needed, and then evaluate the mathematical operations, returning the result from each rule, and finally printing it at the end of the start rule (top). Finally, the program was tested. Each possible operation and number type was tested, as well as combinations of operations and parenthesized expressions which change the order of operations.

The final parser worked as expected. Using ANTLR provided an opportunity to get familiar with a different type of parser generator. This helped me develop a better understanding of how to generate a grammar compatible with an LL(*) parser.