Daniel Aldama

CPSI38303

November 23, 2025

# Assignment 1.1 Report

The goal for this assignment is to design and implement a Reverse Polish Notation (RPN) calculator that could accurately assess a couple of arithmetic expressions. This calculator doesn't just handle basic operations like addition, subtraction, multiplication, and division, but also supports trigonometric functions like sine, cosine, etc. Also, the assignment required the ability to correctly process grouped expressions using parentheses, ensuring the proper order of operations (CFI Team). To make this happen we were expected to use standard tools for syntax analysis and parsing, specifically Lex and Yacc, this helped by breaking down and interpreting the input expressions accurately. Overall, this task helped us understand how a functional RPN calculator operates while using programming skills.

. When implementing anything on the RPN calculator, the Lex/Yacc toolset was chosen because it is effective in handling lexical and syntactic analysis. Lex is used to tokenize input expressions. This involves breaking down the input into meaningful components such as numbers, operators, and parentheses. While the process of automating inputs tokenizes, Lex lessens the complexity of identifying the elements, thus making implementation more organized and less flawed. Yacc was used to parse the token stream from Lex. It lets you define grammar rules that describe how numbers, operators, and functions fit together. This approach fits well with the programming paradigm, where the focus is on defining what the grammar looks like rather than how to evaluate it. Yacc handles operator precedence and associativity for you,

meaning Yacc automatically decides the correct order in which operations should be assessed, so you don't have to manually program how expressions like mixed or nested operators. The calculator rates expressions using a stack, which is standard for a Reverse Polish Notation calculator. By letting Lex handle the tokens, Yacc handle the parsing, and C functions hand the actual calculation, each part of the project has a job, so it isn't just one part doing all the work. From all these tools being used together it makes the expression easier to implement. For example, a student could simplify calculations by reducing an expression that contains all four common binary operators to only be able to use addition and subtraction (Vanderbeek).

Now starting with the implementation of the RPN calculator, the lexer identifies numbers, operators, and functions, while the parser uses grammar rules and a stack to make sense of RPN and grouped expressions. Furthermore, actual calculations don't occur until they reach the rpn.c and main.c which provides a simple interface for entering expressions and seeing results. Putting all these components together maintains a specific part of the process.

When using Lex and Yacc, it created a structured workflow for handling the different stages of processing RPN input. Lex generated tokens for numbers, operators, functions, and parentheses, which reduced vagueness when passing information to the parser. Yacc used the grammar rules on the tokens to check if the input was a correct RPN expression. This division made it possible to detect issues such as incomplete expressions, unmatched parentheses, or misplaced operators during parsing rather than during computation.

Supporting features like trigonometric functions and grouped expressions require extending both the token set and the grammar. Nested expressions introduced the need for recursive evaluation so that each parenthesized expression could be processed as a self-contained unit before being used in a larger computation. The stack structure used for RPN worked

consistently with this approach, since each evaluated value could be pushed back for later operations. Throughout the implementation, the interaction between the lexer, parser, and evaluator showed how tools based on formal language principles can be applied to build small interpreters and how grammar definitions influence the flow of evaluation.

Overall, this assignment showed how lexical analysis, parsing, and expression evaluation can work together to build a functioning RPN calculator. Using Lex and Yacc made the structure of the program more organized, since each tool handled a specific part of the process. Once the grammar and tokens were defined, extending the calculator with features like trigonometric functions or grouped expressions became more manageable. The result meets the requirements of the assignment and demonstrates how compiler tools can be applied even to relatively small problems. This assignment also provided useful experience in thinking about expressions the way a parser would, which is helpful for understanding larger concepts in language processing.

References

CFI Team. “Reverse Polish Notation (RPN).” *Corporate Finance Institute*, 22 Nov. 2023, corporatefinanceinstitute.com/resources/data-science/reverse-polish-notation-rpn/.

Vanderbeek, Greg. “Digitalcommons@university of Nebraska - Lincoln.” *DigitalCommonsUniversity of Nebraska Lincoln*, 2007, digitalcommons.unl.edu/.