**CS 303 Data Structures**

Project 1

Lauren Magee and Ben Teig

**Assumptions**

Together we chose to implement Project 1A, to create an infix expression parser using stacks. While thinking about the process to this solution, we made a few assumptions not listed on the project guidelines paper. First we thought because we’re performing mathematical operation(s) we’re going to need to separate out the initial string given in the evaluator function. We did this by splitting up the string into individual characters in a char array. Our second assumption is that we would need more than one stack to complete the parser. We decided that we would use two stacks, one for the operand, or numerical values, and one for the operators in the expression.

**Algorithm**

1. Define two stacks, one for the operands and the other for operators
2. Create Evaluator Function for expression, change string to an array of characters
3. After array is created, check for any expression errors including: divide by zero, having a closing parenthesis or binary operator as the first character, or having a binary operator followed by a unary one
4. If it doesn’t pass through error checks, end program
5. If it does pass through error checks call on stack functions
6. Push values into the appropriate stack
7. Push operands into the appropriate stack
   1. Check if the operator is a special case operator (such as a unary or double character operand) and evaluate accordingly. If not, continue.
   2. If the operator being pushed is of lesser precedence than the operator on the stack, evaluate the last two operands from the stack operator and push the result to the operand stack. Then push the current operator. If precedence is not an issue, push the operator to the stack.
8. After all values are pushed in and the array is “empty”, take out one operator.
   1. If the operator is a unary operator, take out one operand value and evaluate it. If it is binary, take out two operand values and evaluate them. The result is then pushed to the operand stack.
9. **Repeat Step 8** until the operand stack is empty.
10. Return the value evaluated from the given expression to user
11. End Program

**UML Diagram**

terms

ops

1

1

**Stack<char>**

**Stack<int>**

|  |
| --- |
| **Evaluator** |
| * expression: string |
| * errorCheck(string exp): int * removeSpaces(string str): string * eval(string exp): int * precedence(char o): int * solve(): int * uOp(char o): int * specialOp(char o): bool |

**Efficiency of Algorithms**

The majority of our functions utilized either O(n) or O(n^2), the specific Big O notation for all of our major functions in the program are commented on in the project. We viewed this as the best way to document those efficiencies. We do think there’s always a way to make code more efficient, in the middle of editing and reviewing our code, we found ourselves removing nested loops to increase efficiency.

**Contribution of Team Members**

We decided to divide the project as follows: Ben Teig took responsibility for creating the stacks, and getting them to perform the required infix operation, along with making the UML diagram and citing the Big O notation of his functions. Lauren Magee took responsibility for creating the evaluator function and error checks, along with writing the project paper and citing the Big O notation of her functions. Separately they both completed their own individual reports.

**References**

The link to it and our code on Github is <https://github.com/bmteig/303_Project1>

262588213843476, raviteja1452. “Stack/ Evaluating an Infix Expression Using Two Stacks (Operand and Operator Stacks).” *Github*, Oct. 2017, gist.github.com/raviteja1452/8632e6460edc05350f8eacab367452b5.