Calculator Design (v1.0)

**Introduction:**

The Calculator performs the requirements laid out in the Calculator System Requirements Document. The following sections will discuss how these requirements are implemented and the flow of the program.

**The Main Function:**

The main function of this program (as usually for any program) simply guides the flow of the rest of the program. The following is a pseudo code representation:

*Loop forever*

*{*

*Ask the user for an equation input*

*Eliminate any whitespaces from the input;*

*If the user typed in the word “quit”*

*{*

*Print the phrase “Quitting…”;*

*Stop the program (stopping the loop);*

*}*

*If the user did not type the word “quit”*

*{*

*Call the “solveEquation” function and print the result.*

*}*

*}*

**The “solveEquation” Function:**

This function is simply used to call the “calculateRPNExpression” function (where RPN stands for Reverse Polish Notation). The “solveEquation” function takes the equation that the user entered as an input. The function exists in case any other functionality to solve the equation is needed. When calling the “calculateRPNExpression” function, a function call to the “convertToRPN” function is passed to it (that is, the result from “convertToRPN” is passed to “calculateRPNExpression”). When calling the “convertToRPN”, the equation that the user entered is passed in as a parameter.

**The “isOperator” Function:**

This helper function is used to determine whether a character is an operator or not. An operator is defined as being one of the following characters: ‘+’, ‘-‘, ‘\*’, ‘/’. These are addition/positive, subtraction/negative, multiplication, and division, respectively. A character is passed in a input and the function will determine whether the character is an operator or not. If it is, it returns the value of “True”, else it returns the value of “False”.

**The “operatorPrecedenceLevel” Function:**

This helper function is used to determine the precedence level of an operator. There are two precedence levels, namely the Addition/Subtraction precedence level and the Multiplication/Division precedence level. The addition (+) and the subtraction (-) operators belong to the Addition/Subtraction precedence level and the multiplication (\*) and the division (/) operators belong to the Multiplication/Division precedence level. The function will take in an operator as input and determine which precedence level it belongs to, and then return the result.

**The “hasLowerOrEqualPrecedence” Function:**

This helper function is used to determine how two operators relate to each other based on their precedence level. The function first finds out the precedence levels of each operator using the **operatorPrecedenceLevel** function, then if proceeds to compare the precedence levels of each operator. The Multiplication/Division precedence level has a higher precedence than the Addition/Subtraction precedence level. If the precedence level of the first operator parameter is lesser or equal to the precedence level of the second operator, the function will return “True”. If the precedence level of the first operator parameter is greater than the precedence level of the second operator, the function will return “False”.

**The “convertToRPN” Function:**

This function converts the equation that the user entered into Reverse Polish Notation (also known as Post-Fix Notation). As a note, valid input characters are all digits from 0 to 9 and the ‘+’, ‘-‘, ‘\*’, and ‘/’ operators. The following is a pseudo code representation:

*Declare a holder for the operand called “****tempOperand****”;*

*Declare a stack to hold the converted post-fix equation called “****postfixStack****”;*

*Declare a stack to hold operators detected in the equation called “****operatorStack****”;*

*Try*

*{*

*Declare a counter called “****count****” to keep track of the current of position the equation;*

*Declare a Boolean variable called “****unaryMode****” to indicate whether a unary operator has been detected;*

*Declare a holder variable called “****unaryOp****” to hold the detected unary operator (if any);*

*For each* ***token*** *in the equation string*

*{*

*If the* ***token*** *is not a valid character*

*{*

*Raise a ValueError;*

*}*

*Else if the* ***count*** *is equal to “0” and the* ***token******is an operator*** *and the* ***operator precedence*** *is on the addition/subtraction level*

*{*

*Set* ***unaryMode*** *to True;*

*Set* ***unaryOp*** *equal to the* ***token****;*

*}*

*Else if the* ***token******is an operator*** *and the* ***tempOperand*** *holder is not empty {*

*If* ***unaryMode*** *is true and* ***unaryOp*** *is equal to the subtraction operator*

*{*

*Push the negative of the integer value of* ***tempOperand*** *into* ***postfixStack*** *(may throw ValueError);*

*}*

*Else*

*{*

*Push the integer value of* ***tempOperand*** *into* ***postfixStack****;*

*}*

*Set* ***unaryMode*** *to False;*

*Clear* ***unaryOp*** *to an empty string;*

*Clear* ***tempOperand*** *to an empty string;*

*Reverse* ***operatorStack*** *and call that new stack “****reversedOperatorStack****”;*

*For each* ***operator*** *in* ***reversedOperatorStack***

*{*

*If* ***token has a lower or equal precedence*** *to the* ***operator***

*{*

*Pop an operator from the* ***operatorStack****;*

*Push that value into the* ***postfixStack****;*

*}*

*}*

*Push the* ***token*** *into the* ***operatorStack****;*

*}*

*Else*

*{*

*Append the string in* ***token*** *to the* ***tempOperand*** *holder*

*}*

*Increment the value of* ***count*** *by 1.*

*}*

*If* ***unaryMode*** *is true and* ***unaryOp*** *is equal to the subtraction operator*

*{*

*Push the negative of the integer value of* ***tempOperand*** *into* ***postfixStack****;*

*}*

*Else*

*{*

*Push the integer value of* ***tempOperand*** *into* ***postfixStack****;*

*}*

*Clear* ***tempOperand*** *to an empty string;*

*Push any remaining values in* ***operatorStack*** *to the* ***postfixStack****;*

*}*

*Catch ValueError*

*{*

*Clear* ***postfixStack****;*

*Add a value indicating an “undefined result” to the* ***postfixStack****;*

*}*

*Return the* ***postfixStack****;*

**The “calculate” Function:**

This helper function is used to provide a result from a binary operation. The function takes two operands and an operator as parameters. The function determines which operator has been passed in and it then performs the corresponding operation of the two operands. In the case of the subtraction and division operations, the operand parameter that was passed in second (operand 2) is subtracted or divided from the operand parameter that was passed in first (operand 1). The result of the binary operation is then returned. In the case of the division operation, if the first operand is being divided by “0”, the function will return an “undefined result” value. Else, it returns the rounded value of the result (i.e. rounding up occurs when the decimal part of the result is greater than 0.5, else it rounds down). Also, if “undefined result” value is detected in either of the two passed in operands, the function will return an “undefined result” value.

**The “calculateRPNExpression” Function:**

This function is used to provide a result from an expression in Reverse Polish Notation (RPN) or Post-Fix Notation. The following is a pseudo code representation:

*Define a stack to hold the RPN expression called “****rpnExpressionStack****” (passed into the function);*

*Define a stack to hold the final result called “****expressionStack****”;*

*For each* ***element*** *in* ***rpnExpressionStack***

*{*

*If the* ***element is an operator***

*{*

*Define a variable to hold the second operand called “****secondOp****”;*

*Define a variable to hold the first operand called “****firstOp****”;*

*Pop the top value of* ***expressionStack*** *and assign it to* ***secondOp****;*

*Pop the next value of* ***expressionStack*** *and assign it to* ***firstOp****;*

***Calculate*** *the binary operation using* ***firstOp*** *as the first operand,* ***secondOp*** *as the second operand, and the value of* ***element***

*as the operation;*

*Push this value into the* ***expressionStack****;*

*}*

*Else*

*{*

*Push the value of* ***element*** *into the* ***expressionStack****;*

*}*

*Return the top value of* ***expressionStack*** *(this is the final result);*

*}*

**The “displayEquComponent” Function:**

This function is used to display the components that make up the expression in the order they were inputted. The function will take the input and call the **calculate** function to determine if it’s an undefined or a valid expression. If the calculate function returns undefined the **displayEquComponent** will display undefine expression. If is not undefined then the function will proceed to examine the expression and determine if each component is either an integer, whitespace, or operator. It will also display their corresponding values.