Project 2

Erik Sklocic

Joseph Jennings

Sonal Sinha

Infix Expression Parser

Design

**Data Structures Used:**

**ArrayList:** We used an array list with the data type StringBuilder to keep track of all the equations, in postfix form, given by the input file. Our variables, givNum and givNum2, are assigned with the values that are returned from evaluatePostFix() that are parameterized using the arraylist of postfix equations.

**StringBuilder:** We utilized four StringBuilders. In the method infixtoPostFix() the first StringBuilder, sb, keeps track of the infix expressions and creates the postfix expression. Our second StringBuilder, in our method comparator(), stores each comparator that is in the given String, line. This is then appended to the queue. Another StringBuilder is implemented in comparator(), this is called str which keeps track of the equations inside the String which is then appended to the array list, arr. Our fourth and final StringBuilder is utilized in the parameters for evluatePostFix(). This StringBuilder stores the equation that needs to be solved. This equation is solved in the following loop which utilizes variable d to store the integer at the character in the StringBuilder.

**Stack:** Our program implements two stacks. The first stack is in infixToPostFix(). This stack stores the operators inside each line, given by the input file. The second stack is implemented in evluatePostFix() which stores all of the numbers and operators inside the given stringbuilder. Variables, val1 and val2, are then assigned to the popped elements from this stack while looping.

**Queue:** We utilize two queues, both of which are for storing boolean operators, such as: greater than, less than, equal to, does not equal and and/or statements. The first occurrence of a queue is in comparator(). This queue holds all the operators in the string. Our variable, givCom is then assigned to these operators which is used to perform the needed operation. The second queue is passed as a parameter for solveAndOr(). The queue is used to store all of the and/or operators in the specified string. These are then used to evaluate the statements.

**Linked List:** We only used LinkedList to implement our Queues.

**List:** Only two Lists are used in the program. This is seen in our method comparator() where the list andOrNums is appended with the numbers before or after an && or ||. This is done by skipping those operators then adding them to a separate queue. Then the program solves the remaining operators and appends the List with the resulting numbers. These numbers are then passed inside a list to solveAndOr() which iterates through the List to get the values and perform comparisons.

**Methods:**

**Main:** Our main parses the input file using FileInputStream and Scanner. It takes each line and stores it in a String variable which is then sent to the infixToPostFix() method. Our main also outputs the result of the comparator() method which calls evluatePostFix and solveAndOr to solve any necessary conditions.

**checkIfOperator:** This method returns the precedence of basic operators such as: plus, multiplication, division, modulo, and power. The return value is an integer that corresponds to the precedence of the given character in the parameters for the method.

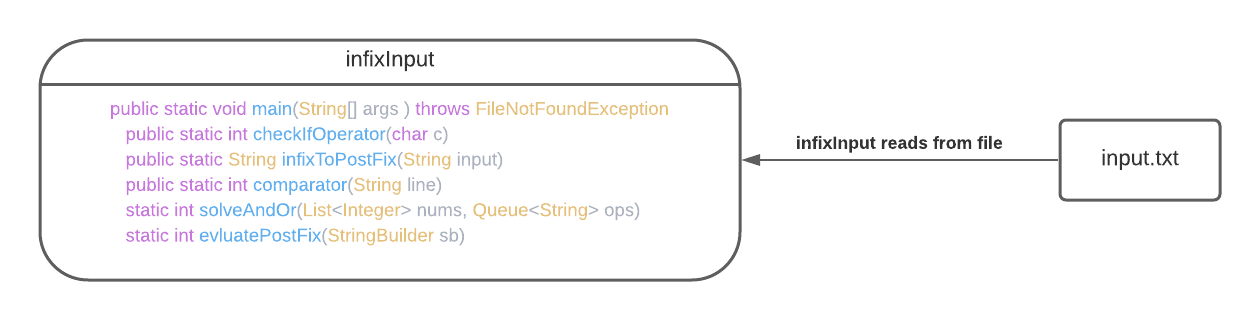
**infixToPostFix:** This method uses a stock to store the operators inside the string, which will eventually evaluate the whole input file. The program then also utilizes a StringBuilder, sb, to continually take the given infix expression and parses it by popping and pushing it when needed. The return value is a completed postfix expression that is stored in that StringBuilder, sb.

**Comparator:** This method takes in a String as an argument. This String is then parsed to its components: operators and numbers. The operators are initially stored in a queue that is named queue. The numbers are appended to a ArrayList<StringBuilder>. These numbers are then checked using the operator in queue by assigning the popped element to givCom, a String. This will then be used to find a result if the operator is not an “and” or an “or”. In the case that it is one of those two, the method will add the operator to a new queue and add the needed numbers to a List of integers. The method returns an integer which will be constantly changed throughout the course of the program.

**solveAndOr:** This method takes in a List of integers and a Queue of strings. The numbers are then compared with the given operator (found in the Queue of strings). This method will return a variable which will store 0 or a 1 depending on if the evaluation was true or false.

**evluatePostFix:** This method takes in a Stringbuilder as its parameter. The method utilizes a stack that will assign two variables, val1 and val2 to the popped values in the stack. This stack is populated by the characters at the specified index, in a loop, found in the StringBuilder. The stack then checks to see what type of operator, then performs that operator on the needed variables. This method alo catches the divide by zero error in which we output a divide by zero output message.

UML



Assignments

**Erik Sklocic:** Assigned to making the infix to postfix function, assisting with additional tasks as needed.

**Joey Jennings:** Assigned to making the boolean expression conditions and parsing/reading the input file.

**Sonal Sinha:** Assigned to making the evaluation of the postfix expressions.

Test Cases

**Test Case 1:**

**Excepted:**

**2%2+2^2-5\*(3^2)**

**-41**

**(1+2)\*3**

**9**

**(4>=4) && 0**

**0**

**1+3 > 2**

**1**

**1==2**

**0**

**2+2^2\*3**

**14**

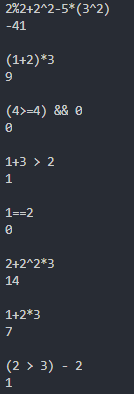
**1+2\*3**

**7**

**(2 > 3) - 2**

**-2**

**Result:**

****

**Test Case 2:**

**Excepted:**

**((2+1)<=1)&&(2 > 0) && (2 < (0 + 3)) && (1==1)**

**0**

**(2>1) && (2 > 1) && (2 > 1) && (2 < 1) && (2 > 1)**

**0**

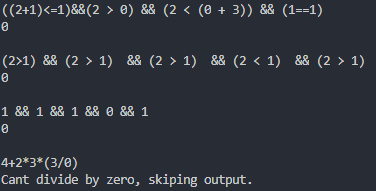
**1 && 1 && 1 && 0 && 1**

**0**

**4+2\*3\*(3/0)**

**Can't divide by 0**

**Result:**

****

Improvements

There could be many improvements that could have been made while working with this project. One of these aforementioned cases is how if you use numbers with more than one character (like 23 or 100) the evaluations would be incorrect. This is due to the fact we are using a string postfix instead of something like a stack, which could have avoided this issue because everything would be pre-separated. Another oversight that could have been avoided is the fact that we treat any string that has a boolean operator in its string as a complete boolean expression and not a part of a larger expression, ie. we cannot perform operations to the return value of a boolean evaluator. This is caused because we removed all of the parentheses after each infixToPost conversion, resulting in a loss of being able to tell the difference between normal equations and boolean equations. For example (2<3)-2 would result in 1 instead of -2 because the code only sees 2<3-2 which is 2<1.