DATA STRUCTURES & ALGORITHMS

A coursework completed as part of the requirement for

SUBJECT NAME: **DATA STRUCTURES & ALGORITHMS**
SUBJECT CODE: **CC105**
LECTURER: **DR. KYAW KYAW HTIKE**

Entitled

ASSIGNMENT TITLE: **ASSIGNMENT 1**

Submitted on

DATE OF SUBMISSION: **3rd AUGUST 2017**

Produced by

Name	Student ID	Course
CHAI JUN SHENG	1001540249	BSc (Hons) Computing
CHEW CHENG JIN	1001540205	BSc (Hons) Computing
JOHN LOH ERN-RONG	1001439193	BSc (Hons) Computing
LOUIS JULIENDO	1001540916	BSc (Hons) Computing
REGINE LIM	1001541495	BSc (Hons) Computing
WAHIDAH BINTI HJ MD DALI	1001334979	BSc (Hons) Computing

Introduction

An arithmetic expression is an expression that produces a numeric value as an output. These numeric values can result in many kinds of numeric values, for example, whole numbers (which is also more widely known as an integer in computer programming) and decimal numbers (also known as double numbers). In data structures and algorithms, there are several notations used to write these expressions. The two different but similar ways of writing expressions which are the Infix expression, and the Postfix expression. An infix expression is the most well-known expression that most people know about, and the usual way of writing expressions in real life. It carries the form of “A *operator* B”, where both A & B are operands of the expression and an operator is written in-between every pair of operands. On the other hand, a postfix expression is an expression that carries the form “A B *operator*”, where an operator is followed after every pair of operands. In a computer’s data structure, the concept of stack can be implemented in the concept of the infix expression and postfix expression, and the conversion between them. A stack is a linear data structure which follows a specific order in which the operations are performed and the order follow the LIFO (Last In First Out) concept.

Background

There is an existence of an algorithm that is being used to convert an infix expression into a postfix expression, which it involves the usage of a stack. However, in the conversion of an infix expression into a postfix expression, the stack is used to hold the operators rather than the operands. The main objective of the stack is to reverse the order of the operators in the expression. One of its functions is to act as a storage structure, since no operator can be printed until both of its operands have appeared.

While it is already easy to perform arithmetic expression based on the infix expression, why does an postfix expression matter and do? This is because as humans, we are probably too used to seeing this form of expression, and so it has become something really easy for a human to understand and process. In the case of computer machines, a postfix expression is a better way for them to process information. With the help of a stack, evaluation of an expression can be done much faster as computer only need to scan operands and operators from the very left to the extreme right. In contrast to this, an infix expression requires a longer time for a computer machine to process, as there is the order of precedence to be taken into consideration. Moreover, any expression can be expressed without the usage of parenthesis. With this format, it is much more suitable for direct execution, therefore decreases any extra processing time for a computer machine to process.

Therefore, in our project, two programs are being developed in our project, namely Infix to Postfix Calculator and Arithmetic Expression Tree Builder, respectively. These programs are developed to help ease the users in converting infix expression, evaluating postfix expression, and drawing a binary tree.

The first program, Infix to Postfix Calculator, is a program that aids in converting an infix expression to postfix expression. The program has an interface in which buttons are displayed. Users can press on the buttons provided to input the infix expression or input numbers and operators that they wish to convert to postfix expression. The program will show the steps of converting the infix expression to postfix expression in a new window, so that users can gain a clearer view on how the algorithm works. Besides, the calculator also includes the function that allows users to evaluate the postfix expression. Similar to conversion of infix to postfix expression, steps will be shown to the users so that users can understand the steps undergone to get the final result.

The second program that is being developed is the Arithmetic Expression Tree Builder. Users can input a new infix expression by selecting the buttons from the interface or using keyboard input. The program will take in the input and draw the expression to binary tree form. Another function that is provided in this program is image saving. It is an optional function, where it allows users to save the binary tree generated to image.

By having these developed programs, users are able to save their time and increase their accuracy and efficiency while doing the job involving the stated functions.

Methodology

Task 1

The first program that is being developed is Infix to Postfix Calculator. Users will be shown an interface with buttons that works like a calculator. Then, users need to enter the infix expression they wish to convert to postfix by pressing the button on the calculator, or pressing the keys from the keyboard. If users accidentally press the wrong button, they can clear the input by clicking “C” button (which indicate clear) or “CA” button (which functions as clear all), or “Backspace” key if they are inputting through the keyboard. “C” button will only clear the most recent input received, while “CA” button will clear the whole input received. Our program uses six types of error validation, which include:

i. Checking is there any input from the user in Input Expression,
ii. Checking if the users press “C” button when there is no Input Expression to be cleared (empty expression to be cleared),
iii. Checking the validity of the infix input, where users are not allowed to enter more than one digit number,
iv. Checking the validity of the infix input, where users are not allowed to enter String data input,
v. Checking the validity of the infix input, where incomplete infix expression can be checked, such as ending with an or multiple operator(s),
vi. Checking if the input expression contains any “/0” (divide by zero) expression, and
vii. Checking whether the Infix Expression is being converted to postfix before evaluating the result.

After entering the infix expression, users need to click “Convert to Postfix” button in the calculator to convert it to postfix. A window, namely “Infix Conversion”, with step-by-step conversion will prompt out, where users can check the steps needed to convert the expression. After the result is shown, users can click “Close” button to continue with evaluation of postfix output. Users will be brought back to the calculator, with the postfix output printed under “Postfix Output” text field. Users will need to click on “Evaluate Postfix” button to do the evaluation. Similar to the previous step, a new “Postfix Evaluation” window with step-by-step evaluation will be shown to the users. Users can close the window by clicking “Close” button, and they will be brought back to the calculator with the result of evaluation being printed. Users can start a new conversion by clicking “CA” button, or leave the program by closing the calculator.

The codes are written in Java programming language which is being run and compiled using the NetBeans IDE.

One of the data structure being used in the infix to postfix conversion is Stack. Stack is used to push, pop or peek elements from the stacks in which can be used for the postfix conversion. Hence, the built-in stack class are imported in our program in order to be used.

Most of the codes are utilizing if-else method to check the condition. The if-else statements are written inside the method. There are 9 methods in total, which are isOperator(), isNumeric(), checkPrecedence(), stackFormat(), stackFormatEva(), Convert(), isOperatorEva(), ConvertEva(), and the last is the checkInput() method.

The isOperator() method has 1 argument which is “operator” as character data-type. It checks the input by using if-else statement to evaluate whether it is an operator. If the input is an operator, it will return true. If the input isn’t an operator, hence, it will return false.

The next method is the isNumeric() method. isNumeric() method contains of one argument, which is “number” as character data-type. This method is used to check if the input is numeric or otherwise, and true will be returned if it is numeric.

The checkPrecedence() method has 2 arguments which are “O1” and “O2” as character data-type. It checks the input by using if-else statement. There are 4 conditions that need to be checked, which are:

a. O1 and O2 are “+” or “-“ respectively
b. O1 is “+”, “-“, “*” or “/” and O2 is “*” or “/”
c. O1 is “+”, “-“, “*” or “/” and O2 is “^”
d. conditions other than the three conditions listed above.

If the conditions from a to c are fulfilled, it will return true, as for condition d, false will be returned.

The fourth method in the program is stackFormat() method. It takes up one argument, which is “s” as String data-type. This method is used to remove all “,”, “ ” and “[]” in the stack, which is default using Stacks classes.

As for stackFormatEva() method, it contains one argument, which is “sEva” in String data-type. This method is applied for the output of the stack in postfix evaluation, where it removes all “,” and “[]” in the stacks.

The Convert() method is for converting the infix expression that is input by the users to postfix expression. This method has one argument, which is the “input”. This argument accepts text from the infix expression text field, and the data-type for it is String.

There is another method named isOperatorEva(). This method has one argument in character data-type, which is the “op”. The method is used to check the presence of operators in postfix evaluation, and will return true if operator is read.

ConvertEva() method is a method to evaluate postfix expression. The argument used is “input”, which is in String data-type. It takes the expression from the infix expression text field and converts it a postfix expression.

The last method is the checkInput() method. It is a method specifically created for error handling. This method has one argument in String data type, named “input”. This method is used to take the input from the infix expression text field, that is the input by the user. This method checks if the scanned character in the text field is even or odd. Even indexes of the expression should only be operands, while odd indexes of the expression should only be operators. This method uses other methods as well, like isNumeric() and isOperator(), where it verifies whether the scanned index is even (which should be an operand) and whether the scanned index is odd (which should be an operator).

Task 2

The second program, Arithmetic Expression Tree Builder, is a binary expression tree drawer. Users will be shown an interface with buttons that works like a calculator, and spaces to draw the binary tree. Then, users need to enter the infix expression they wish to convert to a binary expression tree by pressing the buttons on the window or inputs from keyboard. After that, a new window called "Expression Tree" will appear and it will generate or show the expression tree of the infix input. There are 3 buttons in the new window. The first button is "New", which allows users to create a new tree by entering a new infix expression. The second button is "Save", where its function is to save the tree expression into .jpg images. The last button is "Exit", which is used to exit the program. Our program uses five types of error validation, which include:

i. Checking is there any input from the user in Infix Expression,
ii. Checking if the users press "C" button when there is no Infix Expression to be cleared (empty expression to be cleared),
iii. Checking the validity of the infix input, where users are not allowed to enter more than one digit number,
iv. Checking the validity of the infix input, where users are not allowed to enter String data input,
v. Checking the validity of the infix input, where incomplete infix expression can be checked, such as ending with an or multiple operator(s), and

The codes are written in Java programming language which is being run and compiled using the NetBeans IDE.

In this program, four classes are used to implement the program. The first class is BTree class. We perform a generic type invocation using the <> to reference the generic BTree class, which replace *Tree* with some concrete value such as String or character. We have an inner class, which is called Node class, resides in BTree class. This Node class has Tree type parameter. We instantiate variable that takes in data and store into Tree type parameter. There are one method and one constructor in Node class. The constructor takes data of Tree type as parameter, and it reads the data and set the left and right node to null. As for the method, toString() method, it is used to convert the data to String.

In BTree class, the constructor available is the default constructor, which is used to define the root as null. The methods in this class includes isOperator() method, where it takes in "ch" as argument. "ch" is of String data-type. This method contains a switch. The function of this method is to check whether if "ch" is any operator, such as "+", "-", "*" and "/". It returns true if any of the cases match, and returns false if it does not match any of the cases.

The next method that is included in this class is the checkPrecedence() method. This method takes in "expression" as an argument that is of String data-type. The purpose of this method is to check the precedence of the operators in an infix expression that is to be input by the user. A total of two loops are included in this method, each to check whether it contains "+" or "-", that is of lower precedence, and "*" or "/", that is of higher precedence respectively.

Another method that is also included in this class is constructTree() method. This takes in "expression" as an argument that is also of String data-type. It assigns the root node with a recursion of another method of the same name, constructTree() that takes in three arguments.

The fourth and final method included in this class is a generic method, called the Node <Tree> constructTree() method. It takes in three arguments that are "expression", "Node <Tree> node" and "space", that are of String data-type, node data-type and integer data-type respectively. This method includes other methods that are invoked outside of this method as well, such as isOperator() and the checkPrecedence() method. The checkPrecedence() method takes "expression" as its argument. The output will be then stored into "o" of String data-type. Then, "o" will be used as an argument in the isOperator() method, to check if it is an operator. If it is an operator, a new node will be created. After that, a new array of Strings named "leftrightchild" is declared to take in a split "expression". The element in index 0 of array "leftrightchild" will be then stored into a declared String called "leftchild" while the element in index 1 of array "leftrightchild" will be stored into another declared String called "rightchild". The left node will then be assigned by a recursion of the same method, and the same applies to the right node.

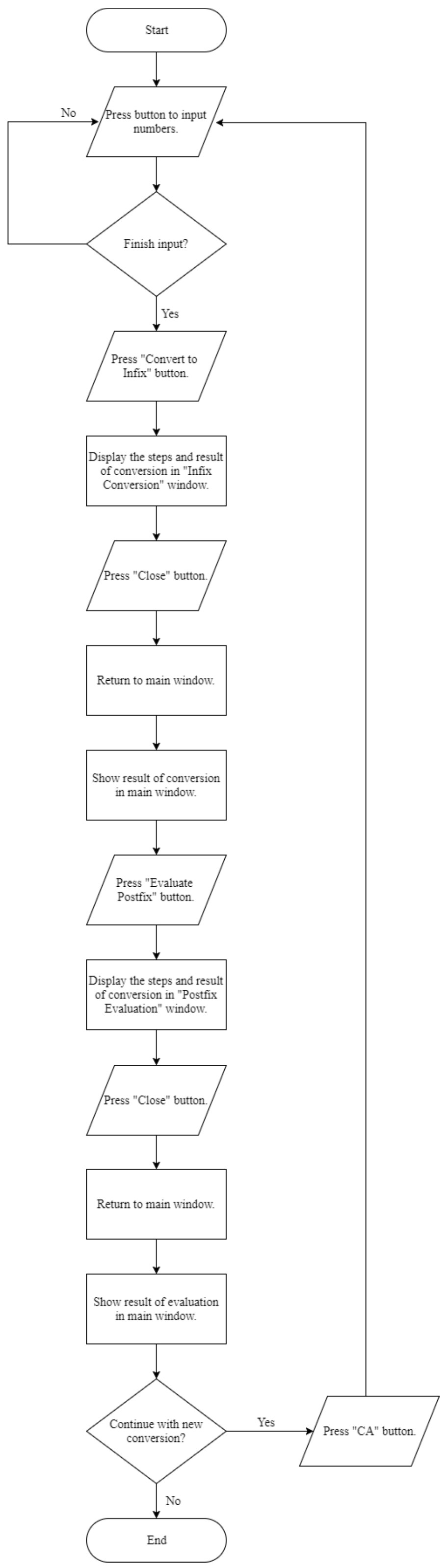
In TreeGUI class, there is an inner class presence in it, which is called the Builder class. This class inherits from BTree class. There is one method in this class, which is the buildTree method. It takes in 5 arguments, which are node, xCoordinate, yCoordinate, space and gg. This method is used to build the tree by giving values to these arguments.

As for TreeGUI class, there is a default constructor which is used to set the size of the frame. The method in this class, setinfixExpression() method takes in infixExpression with String data-type as input. The other method includes paintComponent(), where the properties of the data to be printed out was being set here.

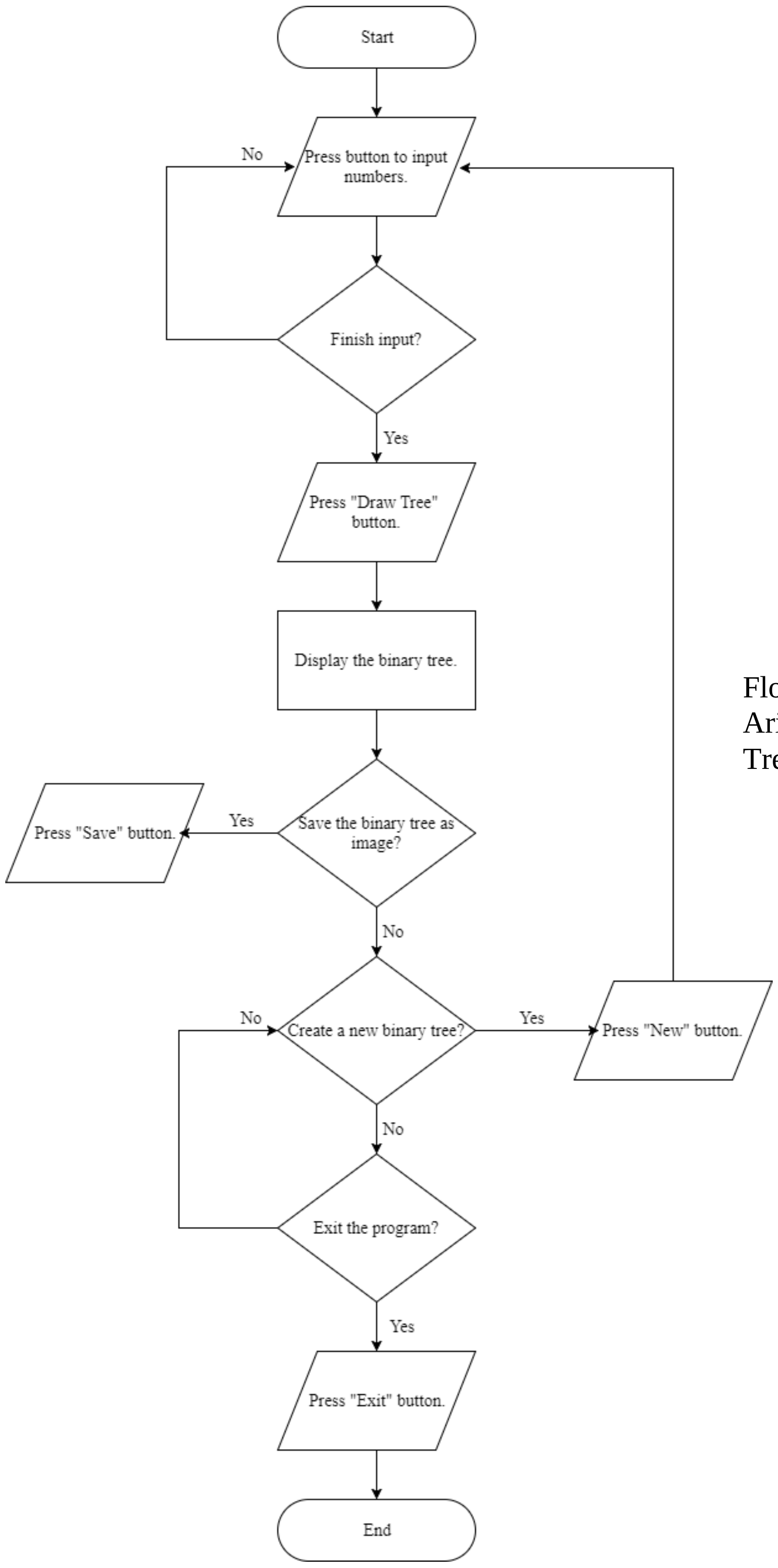
The next class is Drawer class. The method that can be found from this class are checkInput(), isOperator(), isNumeric() and init(). The special method here is init() method, where it extends JApplet. Every extended JApplet class must have one init() method. It is easy to implement the init() method because it is easy to create JFrame frames and JPanel panels. Also, it is easy to add in Buttons, Text Fields and Labels into the panels.

There is also three inner classes in this Drawer class, which are the ButtonListener, NumberListener, and ExitListener class. ButtonListener listens to the action of the button to trigger the events accordingly. As for Number Listener, its function is getting the input from the button when user pressed it. Lastly, ExitListener is used to exit the program.

Flowchart


Start
Press button to input numbers.
No
Finish input?
Yes
Press "Convert to Infix" button.
Display the steps and result of conversion in "Infix Conversion" window.
Press "Close" button.
Return to main window.
Show result of conversion in main window.
Press "Evaluate Postfix" button.
Display the steps and result of conversion in "Postfix Evaluation" window.
Press "Close" button.
Return to main window.
Show result of evaluation in main window.
Continue with new conversion?
Yes
Press "CA" button.
No
End

Flowchart for Infix to Postfix Calculator



Flowchart for Arithmetic Expression Tree Builder

Pseudocode

Task 1

a. Converting Infix Expression to Postfix Expression

```
BEGIN
INPUT infix expression
IF infix expression is NOT empty
    IF infix expression is numeric AND has operator THEN
        INITIALIZE num stepPostfix as 0
            FOR i = 0 TO length-1 step 2 DO
                ASSIGN char at i TO charRead
                INCREMENT stepPostfix
                    IF charRead is operator THEN
                        WHILE check precedence for charRead and peek
                        from stack s
                            ADD output and pop from stack s
                            STORE TO output
                        END WHILE
                    WRITE charRead TO stack s
                    ELSEIF charRead is "(" THEN
                        WRITE charRead TO stack s
                    ELSEIF charRead is ")" THEN
                        WHILE peek from stack s is not "("
                            ADD output and pop from stack s STORE
                            TO output
                        END WHILE
                    IF peek from stack s is "("
                        REMOVE from stack s
                    ELSE
                        ADD output and charRead
                        STORE TO output
                    END IF
                PRINT  "Step ", stepPostfix
                PRINT "Character read: ", charRead
                PRINT "Stack: ", s
                PRINT "Output String: ", output
                END FOR
            WHILE stack s is not empty
                ADD output and pop from stack s
```

```
            STORE TO output
        END WHILE
    RETURN output
END
```

b. Evaluation of Postfix Expression

```
BEGIN

READ postfix expression

    INITIALIZE num outputEva as 0

    INITIALIZE num stepEva as 0

        FOR i = 0 TO length-1 step 1 DO

            ASSIGN char at i TO charReadEva

            INCREMENT stepEva

            INITIALIZE num a as 0

            INITIALIZE num b as 0

                IF charReadEva is not operator THEN

                    WRITE charReadEva to stack sEva

                ELSE

                    ADD pop from stack sEva to b

                    ADD pop from stack sEva to a

                SWITCH charReadEva

                    CASE “+”

                        ADD a and b

                        STORE TO outputEva
                        WRITE outputEva to stack sEva
                    BREAK
                    CASE “-”
                        MINUS a and b
                        STORE TO outputEva
                        WRITE outputEva to stack sEva
                    BREAK
                    CASE “*”
                        MULTIPLY a and b
                        STORE TO outputEva
                        WRITE outputEva to stack sEva
                    BREAK
                    CASE “/”
                        DIVIDE a and b
                        STORE TO outputEva
                        WRITE outputEva to stack sEva
                    BREAK
                    CASE “^”
                        POWER a and b
```

```
                STORE TO outputEva
                WRITE outputEva to stack sEva
            BREAK
            DEFAULT
                PRINT “Error”
        END SWITCH
        END IF
IF charReadEva is not operator THEN
    PRINT “Step ”, stepEva
    PRINT “Character read: ”, charReadEva
    PRINT “Stack: ”, sEva
    PRINT “Operation: ”, outputEva2
ELSE
    PRINT “Step ”, stepEva
    PRINT “Character read: ”, charReadEva
    PRINT “Stack: ”, sEva
    PRINT “Operation: ”, a, “ ”, charReadEva, “ ”, b, “=”,
    outputEva
END IF
```

```
    END FOR

RETURN outputEva

END
```

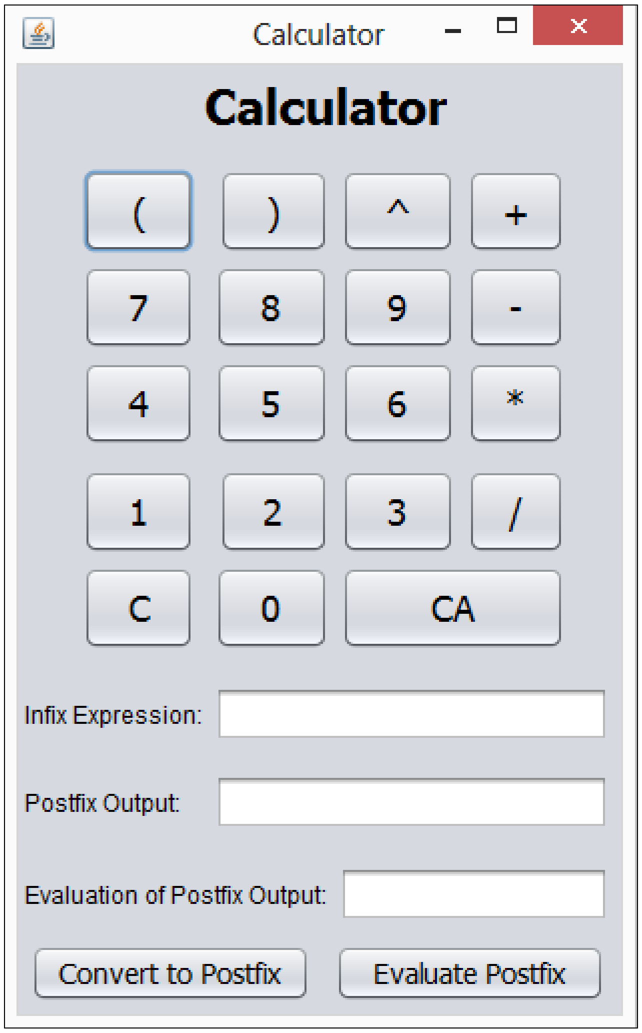
Result

Task 1

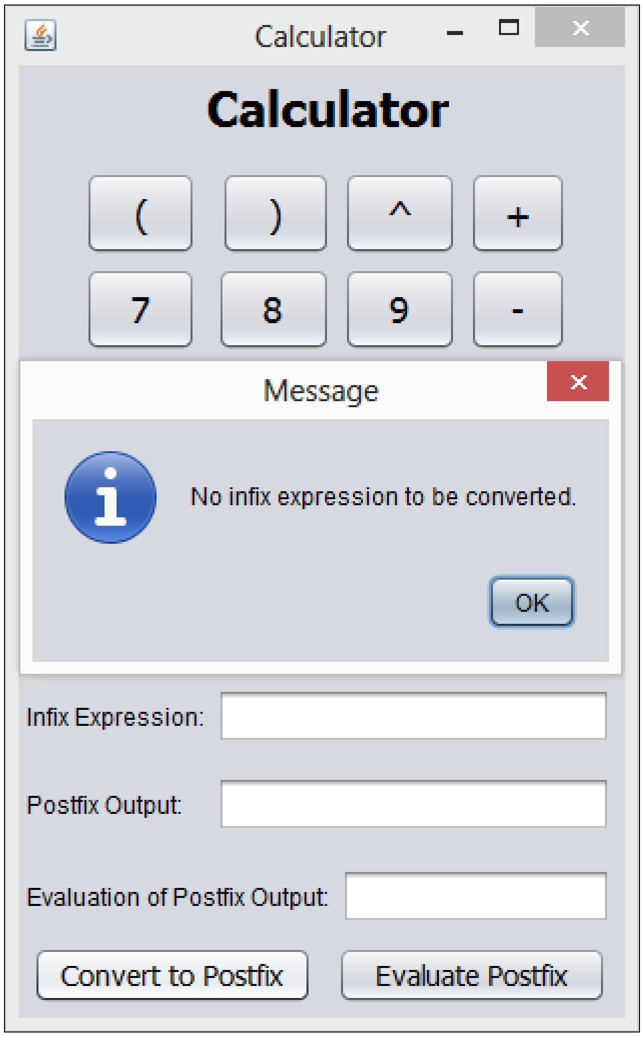
The features of our first program, which is Infix to Postfix Calculator are as below:

a. Converting infix expression to postfix expression
b. Evaluating postfix expression
c. Error validation

Screenshots of the program

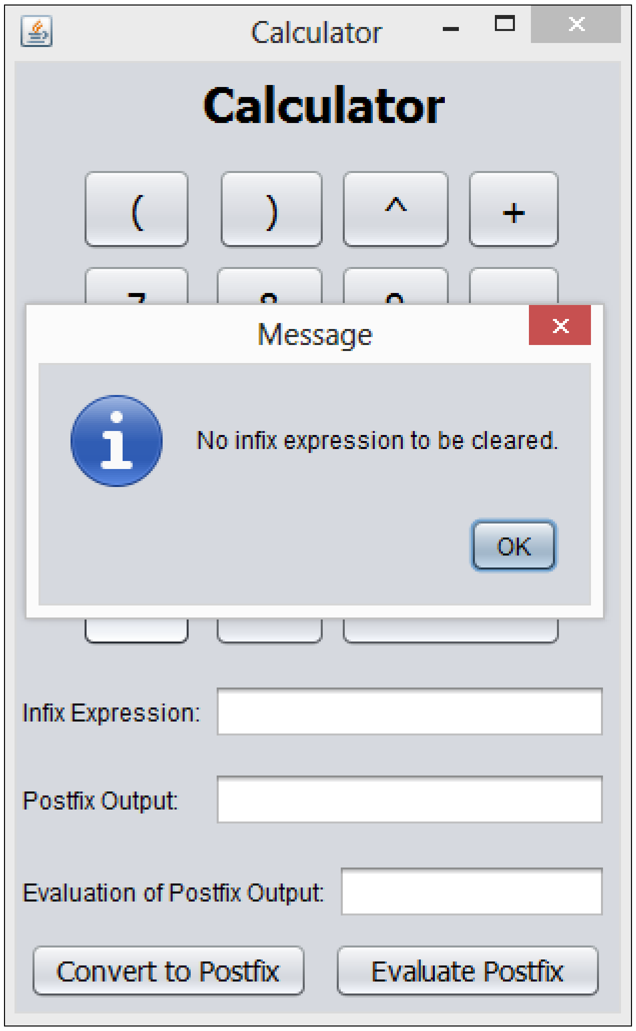


- The interface of the program developed to convert infix expression to postfix output and evaluate the postfix output.



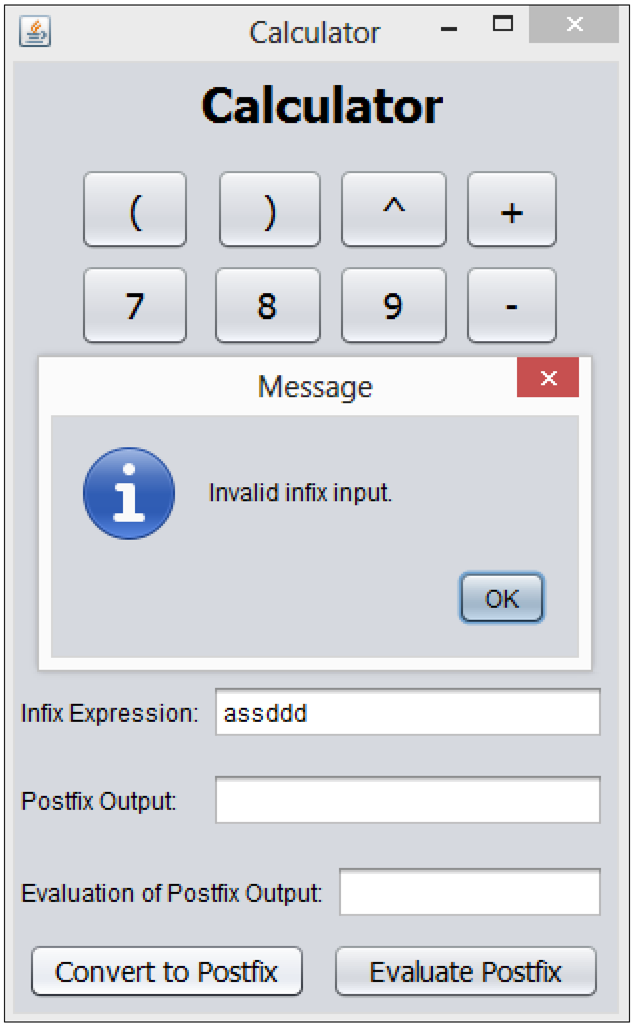
No infix expression to be converted.

- Error handling in which error message will prompt out if users click “Convert to Postfix” button when no infix expression is entered.



No infix expression to be cleared.

- Error message that will pop out if the users click on clear button ("C") or clear all button ("CA") when the Infix Expression text field is empty.



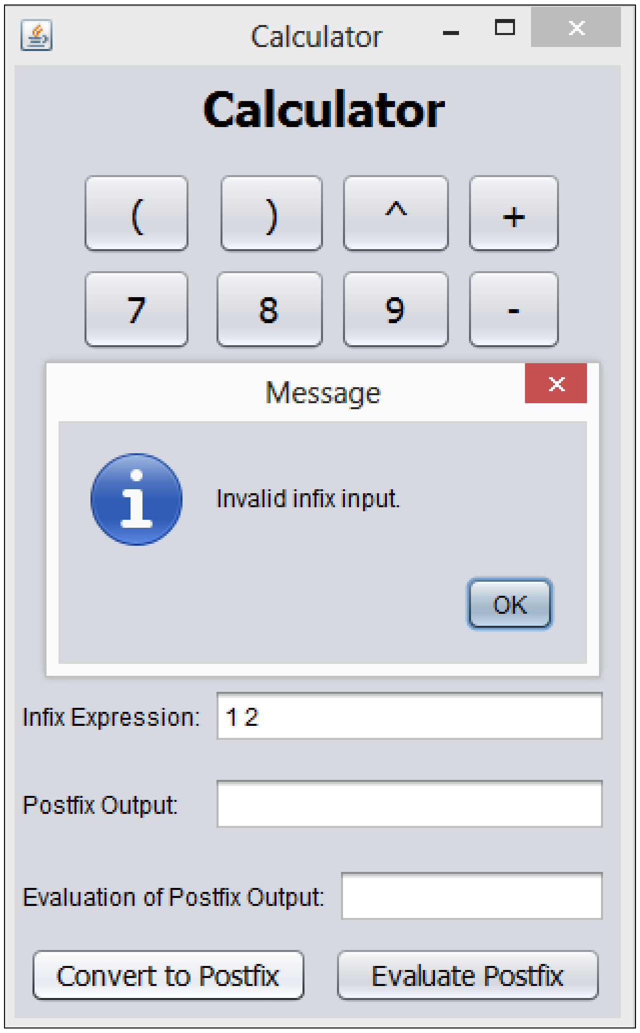
Invalid infix input.

Infix Expression: assddd

Postfix Output:

Evaluation of Postfix Output:

- Error message will prompt out if users enter String data instead of integer data.



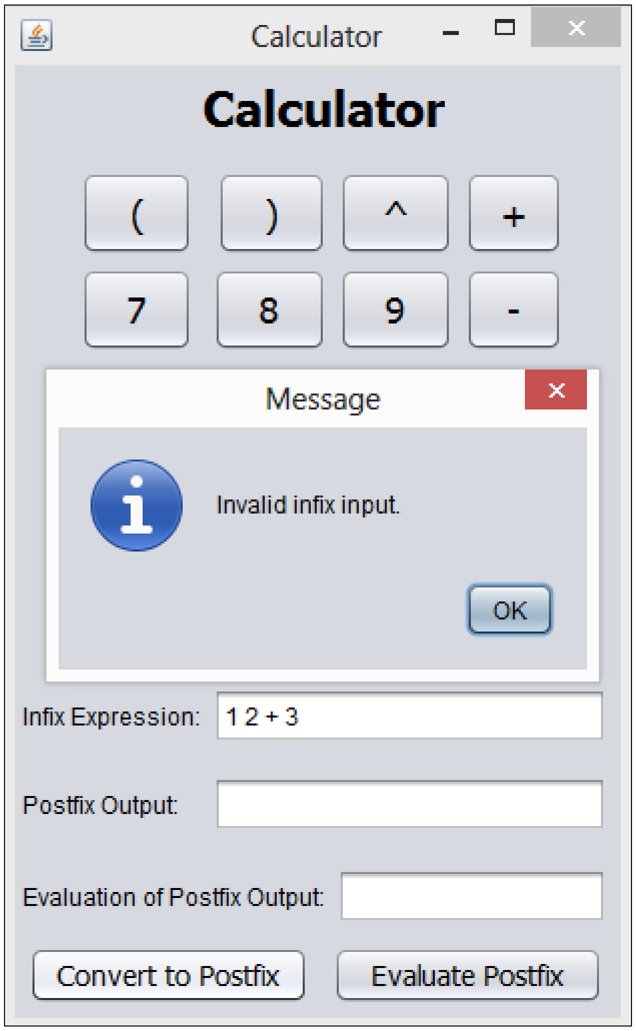
Invalid infix input.

Infix Expression: 1 2

Postfix Output:

Evaluation of Postfix Output:

- Error handling where error message will pop out when the infix input is invalid for conversion – no operator in the expression.



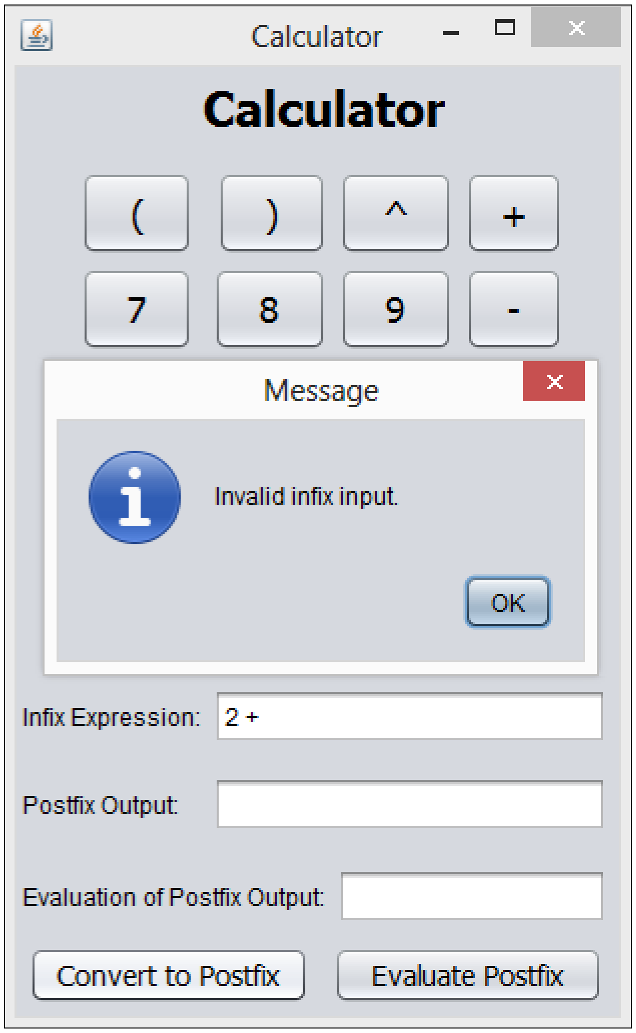
Invalid infix input.

Infix Expression: 1 2 + 3

Postfix Output:

Evaluation of Postfix Output:

- Error handling with error message when users try to input 2 digits value for infix expression – users are only allowed to enter one digit value.



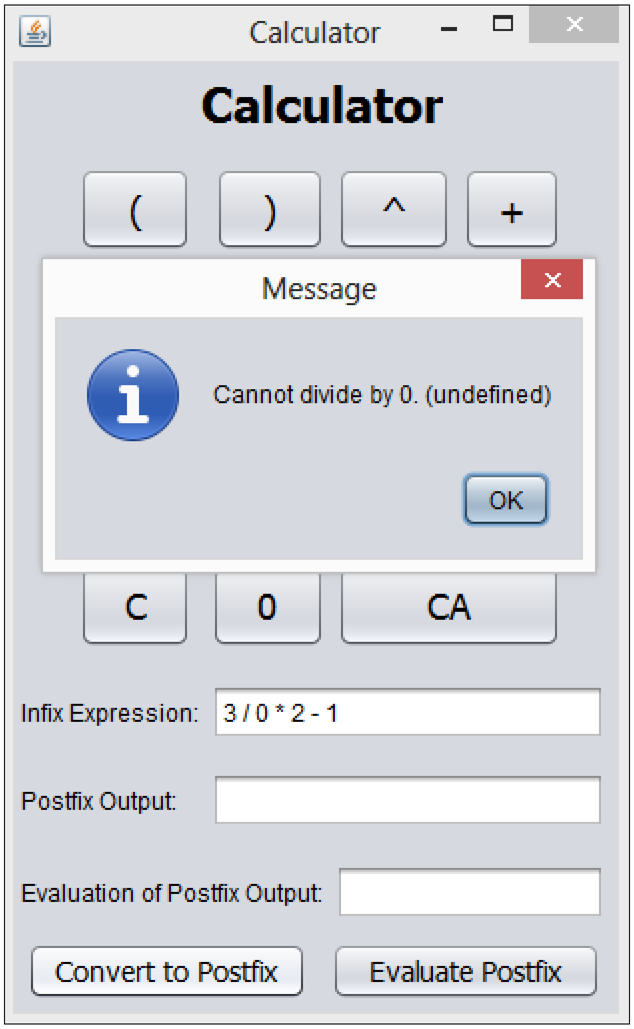
Invalid infix input.

Infix Expression: 2 +

Postfix Output:

Evaluation of Postfix Output:

- Error message shown when the infix input is invalid – users cannot end the expression with operator.



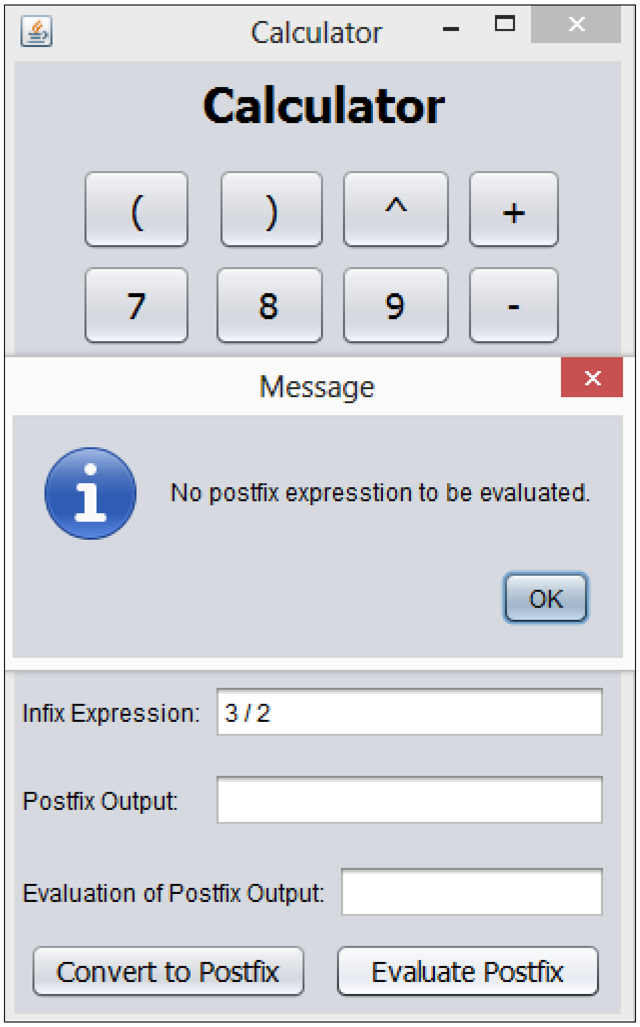
Cannot divide by 0. (undefined)

Infix Expression: 3 / 0 * 2 - 1

Postfix Output:

Evaluation of Postfix Output:

- Error message where user enter infix expression that need to be divided by 0.



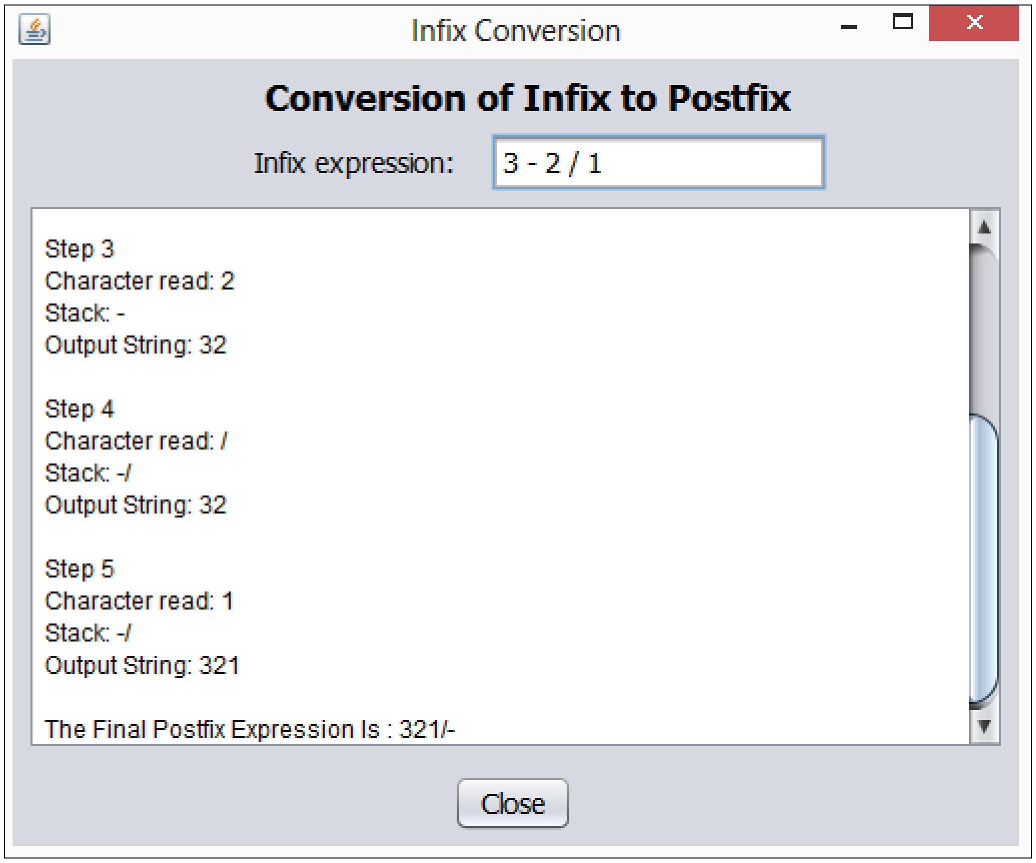
No postfix expresstion to be evaluated.

Infix Expression: 3 / 2

Postfix Output:

Evaluation of Postfix Output:

- Error message prompting out when users choose to evaluate postfix but postfix conversion is not done yet.



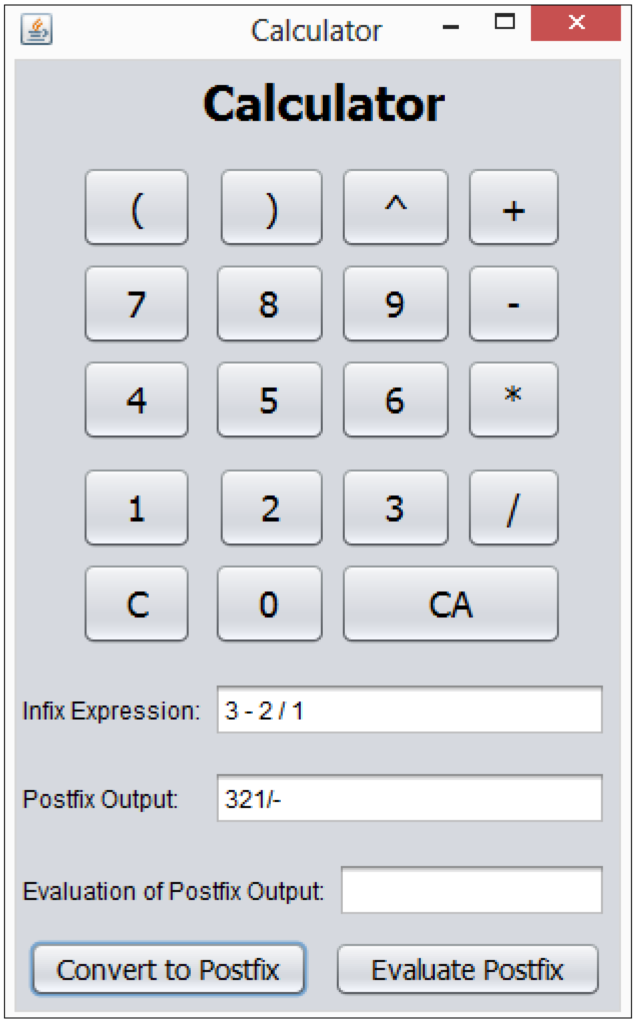
```
Step 3
Character read: 2
Stack: -
Output String: 32

Step 4
Character read: /
Stack: -/
Output String: 32

Step 5
Character read: 1
Stack: -/
Output String: 321

The Final Postfix Expression Is : 321/-
```

- Steps showing the ways to convert the infix expression.

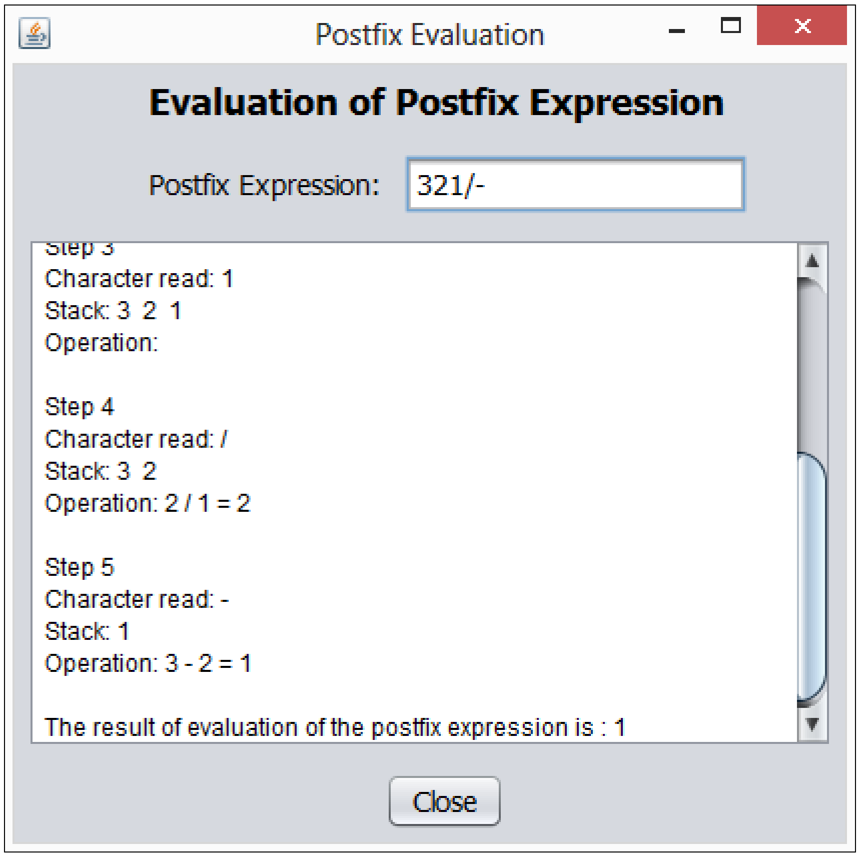


Infix Expression: 3 - 2 / 1

Postfix Output: 321/-

Evaluation of Postfix Output:

- Postfix output will be shown in the calculator.



```
321/-
```

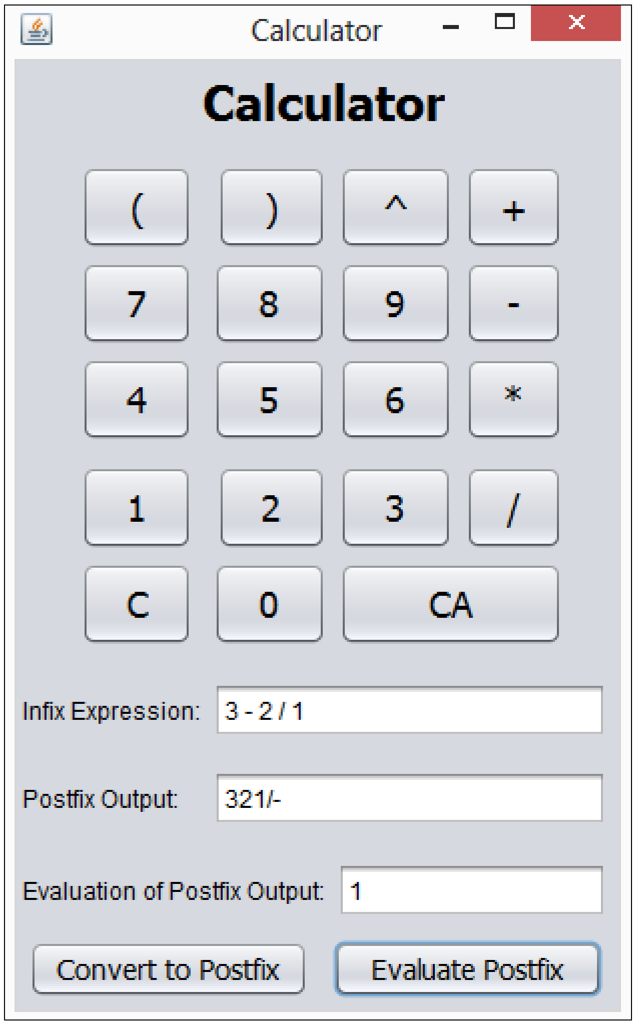
```
Step 3
Character read: 1
Stack: 3  2  1
Operation:

Step 4
Character read: /
Stack: 3  2
Operation: 2 / 1 = 2

Step 5
Character read: -
Stack: 1
Operation: 3 - 2 = 1

The result of evaluation of the postfix expression is : 1
```

- Step by step method showing evaluation of postfix expression.



Infix Expression: 3 - 2 / 1

Postfix Output: 321/-

Evaluation of Postfix Output: 1

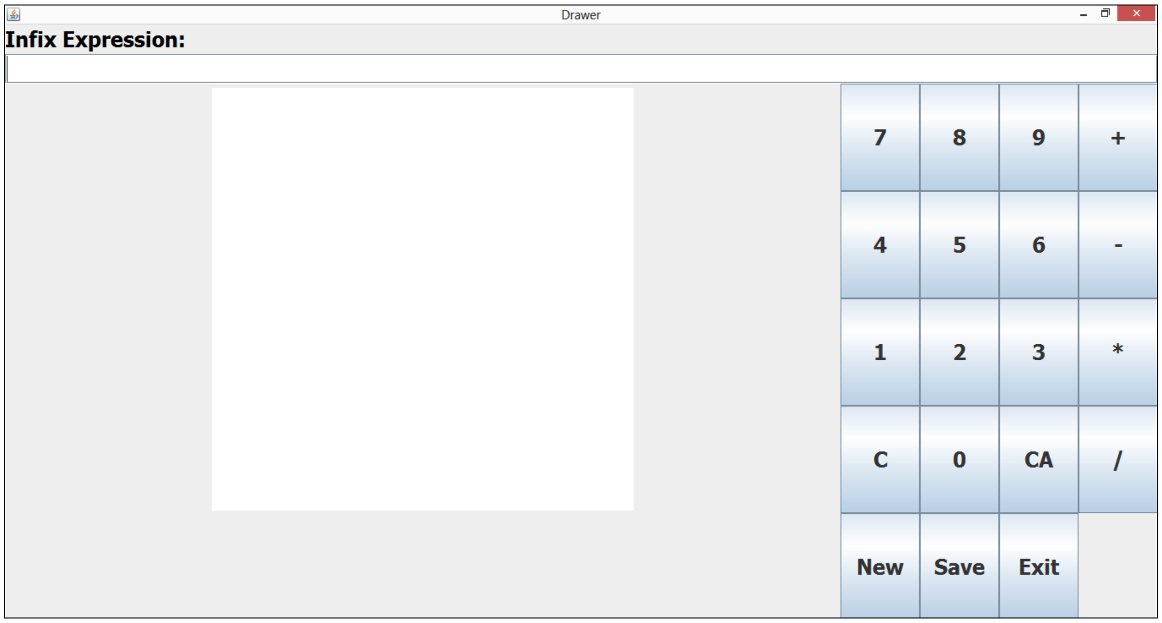
- Result of evaluation will be shown on calculator.

Task 2

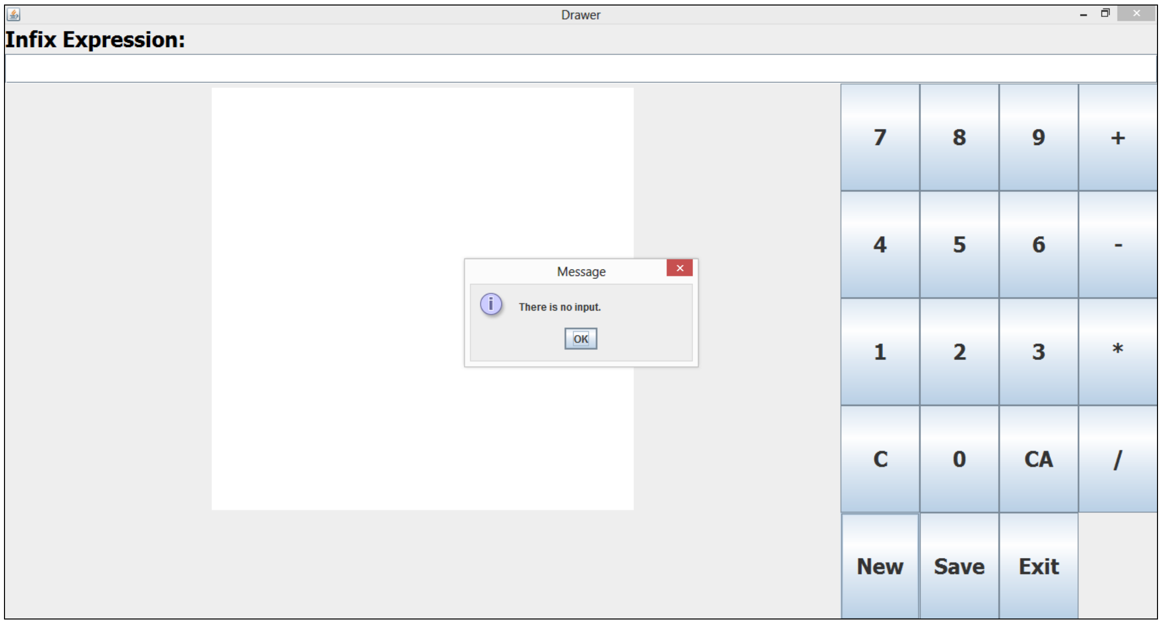
The second program, Arithmetic Expression Tree Builder has the following features:

a. Converting infix expression to binary tree
b. Saving image of the binary tree
c. Error validation

Screenshots of the Program

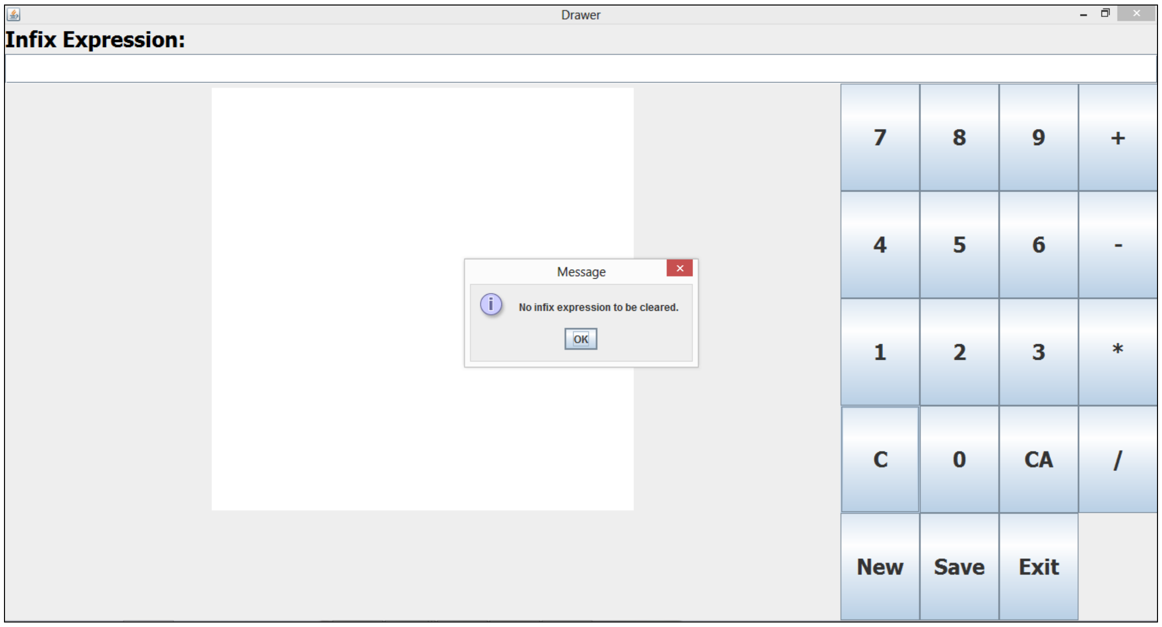


- The interface is designed to help the users to enter the calculation and draw a binary tree.



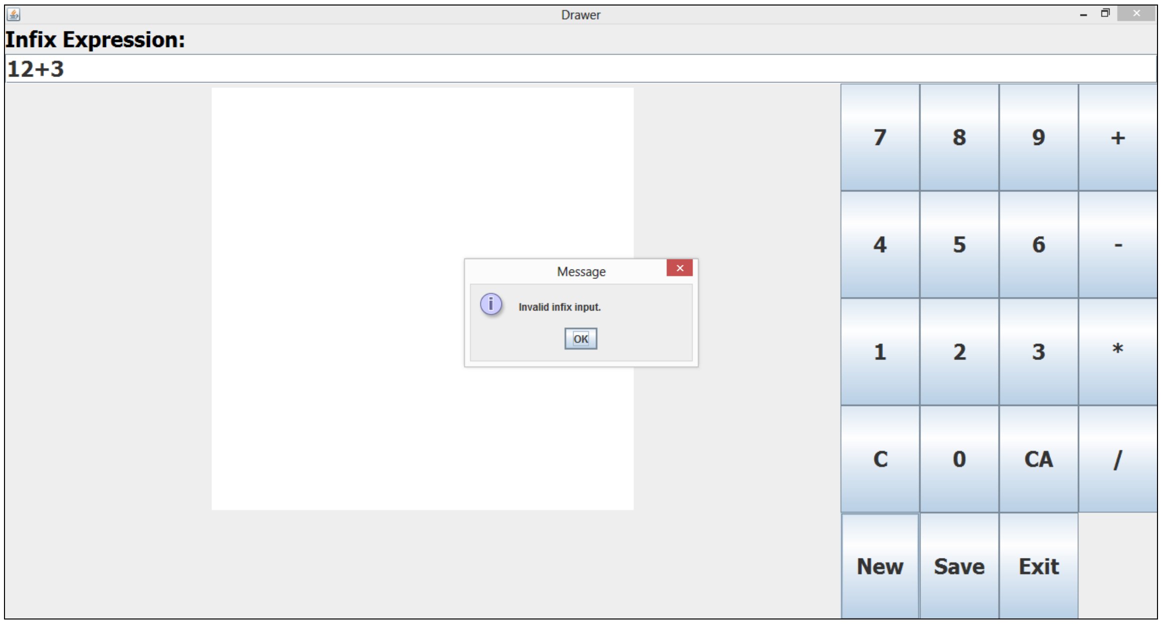
There is no input.

- Error validation when no infix expression is detected from the users.

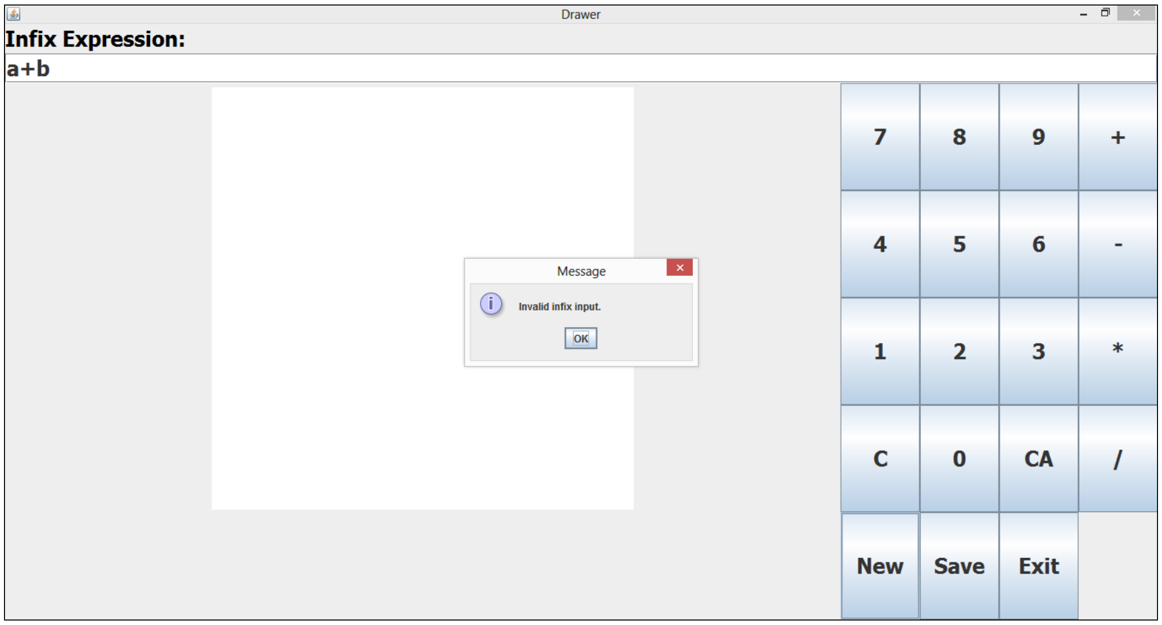


No infix expression to be cleared.

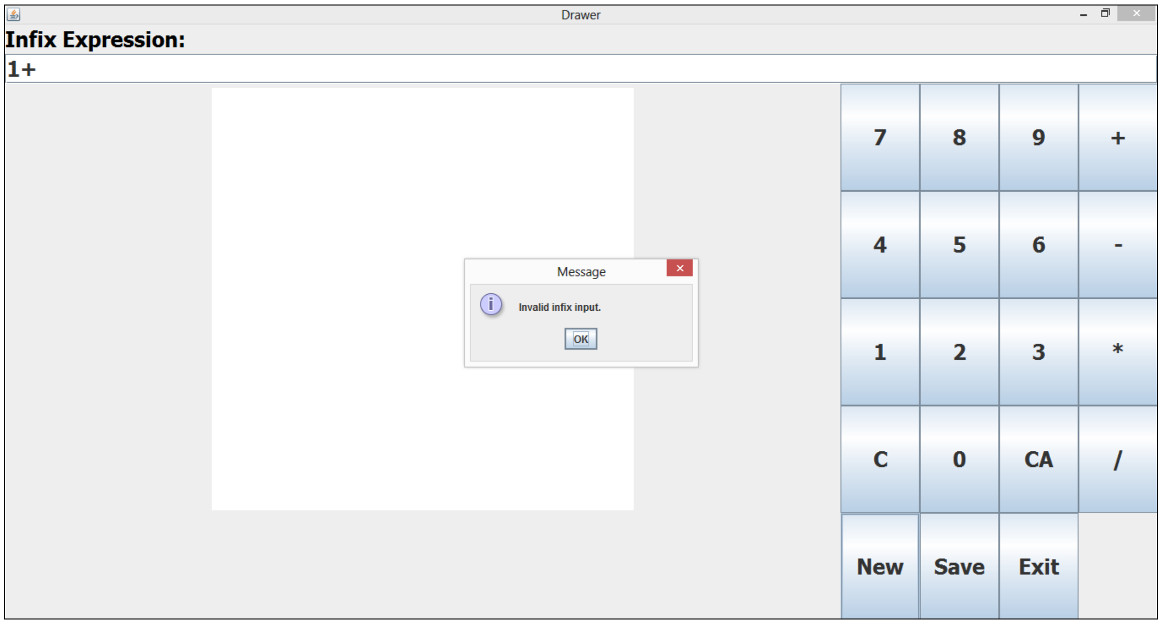
- Error validation when users want to clear an empty infix expression.



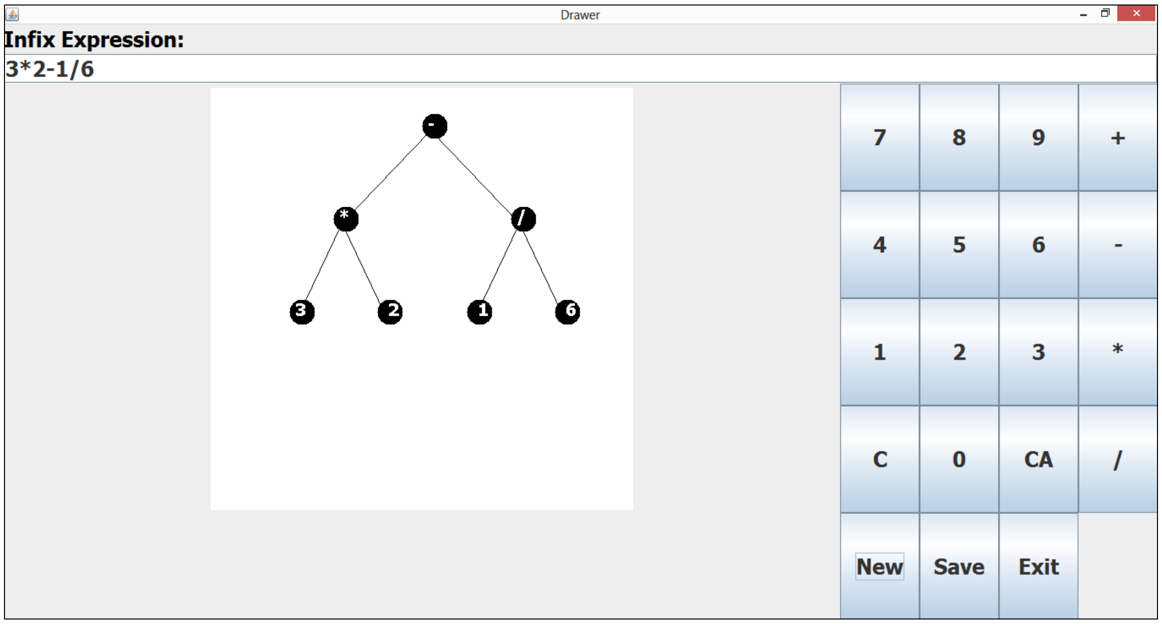
- Error validation where users are not allowed to enter more than one digit number in infix expression.



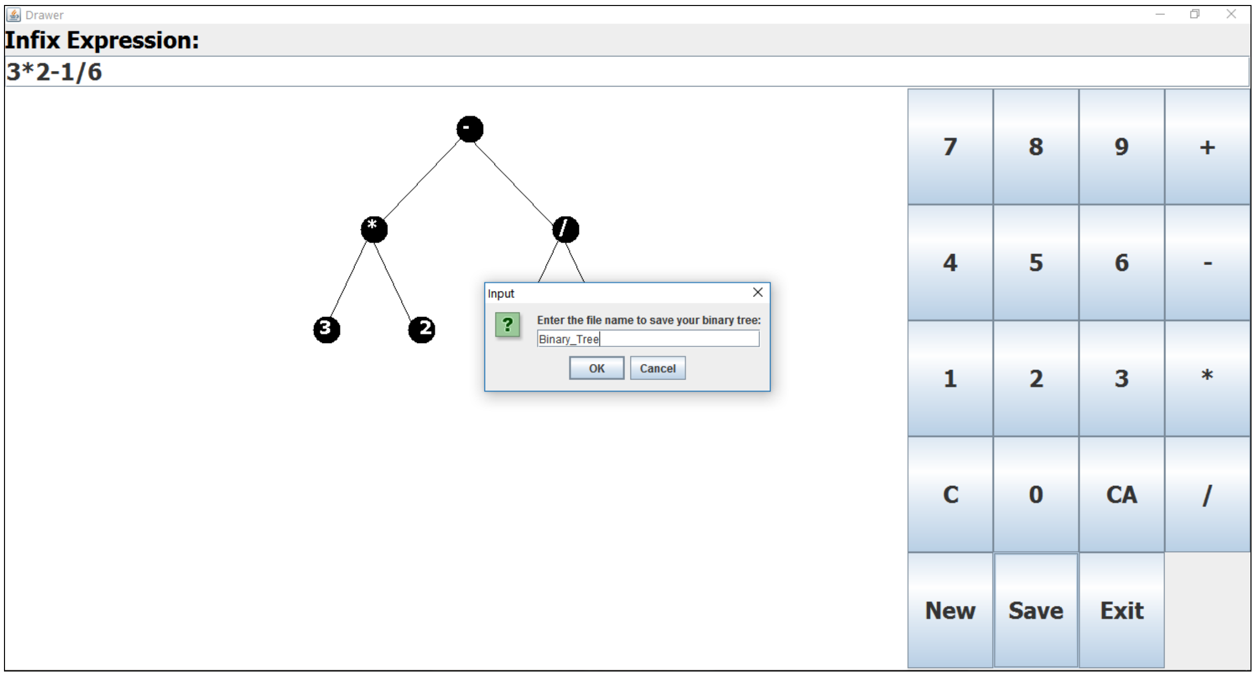
- Error validation where users are not able to enter String data for infix expression – only numeric value is accepted.



- Error validation where incomplete expression entered by the users can be detected.



- A binary tree drawn from the infix expression input by the users.



Drawer

Infix Expression:

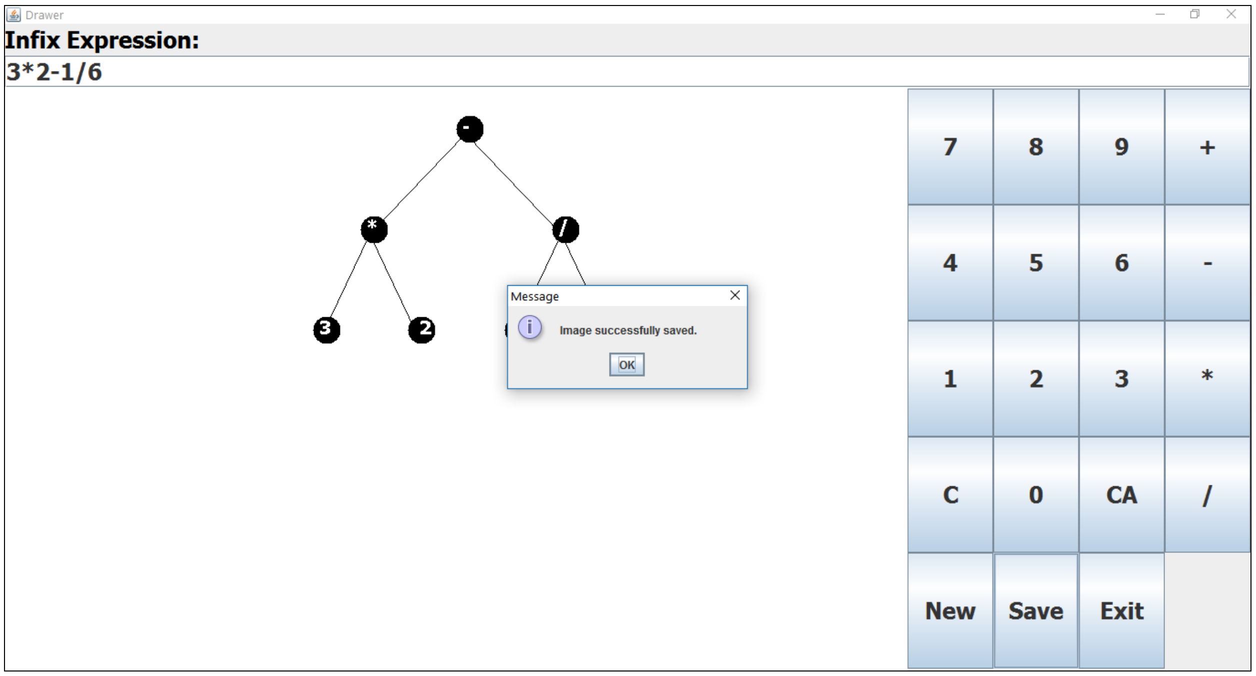
3*2-1/6

Input

Enter the file name to save your binary tree:

Binary_Tree

- Users are able to name the image according to their needs.



Drawer

Infix Expression:

3*2-1/6

Message

Image successfully saved.

- A notification will prompt out to notify users that the image has been successfully saved.

Discussion

1. Strength of the program

The biggest strength of our program is that both the programs that are being developed are easy to use, and the design is simple and straight to the point. Users with or without experience can understand how to use our system easily by just having a look at our design.

The strength for our first program, Infix to Postfix Calculator is that the multifunctional property of the program. In this one program, users are able to perform two actions, which are:

i. Conversion of infix expression to postfix expression, and
ii. Evaluation of postfix expression.

We evaluate our program using Nielson's Ten Heuristics Rules, and found out that our program match itself with a number of rules stated, making it the strength of our program. Firstly, our program matches with the real world. The design of our program is inspired by real-life calculator, where the number buttons is arranged at the similar location as the calculator. The language used in the program is basic English, and it is understandable by the users. The next condition that matches our program is user control and freedom. Users are free to remove any wrong input by pressing "C" or "CA" button from the calculator, or even pressing the "Backspace" key from keyboard.

Other than that, our system fulfils the condition of consistency and standard. Our design of the calculator is also consistent, where the buttons in the calculator are all of the same size. The colour of the buttons is also consistent, so users will not feel confused when using our system. The next rule that we fulfil is helping users recognize and recover from errors. Error validation is present in our program, where it can check various types of error that users might make when using the system. An error message will be prompted out to notify the users when there is error detected by the program.

As for our second program, Arithmetic Expression Tree Builder, the program is easy to use by the users. Similar to our first program, the appearance of this program is similar to a real-life calculator. This program also accepts input by users from the keyboard, as this function actually ease the users. Our second program is designed larger than the first image, as drawing the binary tree takes up larger space compare to the first program. This also fulfills our aim to help the users to have a comfortable experience when using our program.

2. Limitation and Future Improvement

The part that needs to be improved under this Infix to Postfix Calculator is the error validation. There are still errors with validating expression which contains parentheses. Although parentheses function is available in the system, the users will be prompted error message even with the infix expression being error-free. We have tried out by disabling several error validations that are present in our program and found out that this problem is due to the presence of the error validation that checks the validity of the infix input. Users are only able to perform infix expression conversion with parentheses to postfix expression when that error validation is disabled.

Another limitation is that if users enter their input using keyboard, the error validation will not be functioning as well as accepting input from buttons. Users are required to enter a space manually between the operators and operands of the expression, for example, to input infix expression “1+2”, users need to input “1 + 2” for the error validation to work. If no space is entered, the conversion of expression will still work, but the outcome will be wrong.

As for the second task, the main limitation is that the program is unable to receive the input of operators from the button. The only way to input operator is by inputting through keyboard.

The other limitation is that there is slightly logical mistake while building the binary trees. We have tried a number of ways, such as using precedence climbing, iterable and iterators. We realize that we are lack of knowledge in these few topics, and we will try and improve it in the coming future by researching more ways for printing an accurate binary tree.

As for now, our image for binary tree can only be saved in .jpg format. This is one of the limitations for us in saving image. We know about this limitation, and we hope that in the near future we will be able to provide more format such as .png and .pdf format for the users to choose when saving binary tree. We hope to provide more formats so that it can ease the users to keep the files as a reference in the future.

As a conclusion, we are aware about all our limitations, and we will try hard to learn more to improve ourselves so that we will be well-versed in programming. We hope that through this valuable experience, it can be useful in our future.

Teamwork

Task 1

For Task 1, which is the infix to postfix calculator program, Wahidah and Regine were in-charge of designing the Graphical User Interface (GUI) of the calculator. After that, Jun Sheng and John wrote the algorithm for the infix conversion in .java class without GUI. After that has been done, the codes were given to Cheng Jin, and then the three of them applied the algorithm to a JFrame form (to the GUI created).

After they have finished applying the algorithm to the JFrame form, Regine and Louis performed and managed the error handling, standardizing variable names, rearranging and indenting the codes to improve readability, and also adding comments to explain some functions in brief.

Next, Louis was in-charge of writing the pseudocodes and the codes for the postfix evaluation, after that he passed the codes to Jun Sheng and Cheng Jin to apply them onto the JFrame form. On the other hand, John and Regine were in-charge of managing the error handling, standardizing variable names, rearranging and indenting the codes, and also adding some brief comments while Wahidah tested out the system further to check whether there are any errors that aren't being handled properly.

Task 2

For Task 2, John, Regine and Louis were in-charge of writing the algorithm for the binary tree class, driver class, and paintComponent class. After they have finished with the algorithm, they coded the program to draw the tree using those 3 classes. Cheng Jin and Jun Sheng helped in identifying, solving any errors or problems that occurred and they write the codes for saving the binary tree as .jpg image. Wahidah helped in standardizing the variables names, rearranging the codes and writing proper JDialog box messages if there errors arose.

After both tasks are finalized, Louis exported the java class into a jar files so it can be executed within a double-click of mouse without the need to run it in NetBeans IDE.

<u>Conclusion</u>

In a nutshell, the usage of many kinds of arithmetic expressions happens in our world today. Infix expressions are being most understood and known about, while postfix expression is a slightly different format of writing an expression. In spite of the difference, the benefits of an postfix expression that can provide and offer truly contributes a lot and has made a huge impact to our society today. For instance, the design of Arithmetic Logic Units (ALU) that exist on our computer processors today adopts the postfix notation, and has been a universally accepted notation for many years. This form of expression has made computers to perform even better for the many years to come. Also, not to mention the faster processing time of an expression can be performed with the postfix expression. The rule of BODMAS, which stands for Brackets, Orders, Division & Multiplication and Addition & Subtraction is not required to apply with the postfix expression. Therefore, it is important for us, especially people in the Information Technology industry to be aware and remember of its significance and contribution to technology.