CT4021
Introduction to Programming Fundamentals

Assignment-2
Complex Calculator Software

Student Name and Id: Michael Thomas s1605827

Date: 22/04/2017

Contents

1. Introduction

The calculator will be able to solve complex expressions such as “(2+3(5+4))^2”, it will use an easy to understand graphical user interface (GUI), something similar to most calculators on the market today. It will also have a history button which will display the last result with the expression in infix notation.

2. Requirement Analysis

2.1.Must Have

2.1.1. Functional Requirements

Being able to solve complex expressions such as “(2*(2+3)*(5/2))^3”. To do this we will need to implement tokenization, the shunting yard algorithm and a reverse polish notation solver. Without these functions it would be impossible to complete the expressions needed to accomplish the task.

2.1.2. Non-Functional Requirements

Performance, how quick the processing times of the calculations are and how fast they are displayed to the user. If the processing times are slow the user could get frustrated about how slow the calculator is and not use it again, so performance is crucial to making a well functioning calculator.

2.1.3. User Interface Requirements

The user interface will need to look like most other physical and digital calculators on the market, this is because of accordance, because most calculators have the same setup and design the user will usually instantly pick up the calculator and use it with no confusion.

3. Methodology

Agile Methodology

Many different methodologies were discussed for the development of the calculator but ultimately the Agile method was chosen due to the flexibility and iterative process as well as the constant feedback that will be received. Respectively a scrum board was created with 4 main epic stories to work from:

"As a user I want to be able to input an expression like x operator y to get a result"
"As a user I want to be able to input complicated expressions such as (2*(3^3)*(72/5))*(e^3)/0.3"
"As a user I want to be able to get the history of my expressions and history on demand"
"As a user I want to be able to press buttons on a virtual calculator"

These epic stories were split into smaller more manageable stories that were worked on as accomplishing these epics would have been too much work all in one. See appendix C for the stories.

Managing iterative versions of the calculator

Due to the nature of the Agile method there were many iterations of the calculator so the maximum amount of feedback could be received to making the best version of the calculator. Most of the testing was done by the programmer at the early stages such as the basic console calculations e.g "2+2", "5/2".

This early stage was just getting the core foundations of the calculator done, the shunting yard algorithm, tokenization and the RPN solver were all made in this early stage and tested by the developer for technical feedback.

Once the first stage was complete and there were no errors in the calculations, a more advanced shunting yard was implemented for the use of brackets so that longer and more complicated expressions could be inputted and solved. This was important to create and test as this was the stage before the implementation of a GUI.

The core programming was done first without the GUI for the ease of testing and debugging the program, once all bugs and errors were fixed the GUI was created and the code attached accordingly to work with the GUI. This version was tested with the public to ascertain if they understood how the GUI worked and determine how they would input expressions as well as their response to the visual appearance of the GUI.

4. Project

4.1.Design

There are 3 main modules in the design of this calculator, Tokenization, Shunting Yard and RPN Solver. Each one of these plays a vital role in making this calculator actually work and able to accomplish the task given.

4.1.1 Tokenization (Lexical Analysis)

"Lexical Analysis is the process of converting a sequence of characters into a sequence of tokens", read from left to right. A Lexical Analysis program is usually referred to as a "tokenizer", "lexer" or "scanner", in this cause it was called "Tokenization". This was used in this project to identify the characters in the expression and assign the relevant information for that character in a class, then the token is stored in a class list. Below is an example of what can be stored in a token.

Example code

```
struct Token {
    enum TokenType {
        Number, Operator, Left_Paran, Right_Paran, Nothing
    } type;

    //For all tokens
    TokenType tokenType = TokenType::Nothing;
    //If it's a number
    double value = 0.0;
    //If it's an operator
    char symbol = ' ';
    int precedence = 0;
};
```

The reason to use tokenization is so the shunting yard can read and understand what each token is allowing so the shunting yard has the ability to process it.

4.1.2 Shunting Yard

This shunting yard process takes the list of tokens and begins the conversion from infix notation to postfix notation (sometimes known as Reverse Polish Notation (RPN)). Postfix notation was chosen over prefix notation as it takes less key strokes and is known to have less operator errors when executed. Each token is parsed one at a time from left to right through this method, once a token has been parsed it is either added to the RPN list or placed in a temporary stack. Once all have gone through the process the RPN solver is called to finally process the expression for a result to display to the user.

Shunting Yard Algorithm (Structured English)

While there are tokens to read:
    Read a token.
    If it's an operator:
        While there's an operator on top of the shuntingYardList with a greater precedence:
            Pop operators from the shuntingYardList onto the rpnList.
        Push the current operator onto the shuntingYardList.
    Else If it's a left bracket, push it onto the shuntingYardList.
    Else If it's a right bracket.
        While there's not a left bracket on top of the shuntingYardList:
            Pop operators from the shuntingYardList to the rpnList.
        Pop the left bracket from the shuntingYardList.
    Else If it's a number push it to the rpnList.
While there's operators in the shuntingYardList, pop them to the rpnList.

Example Code

```
//if its a right bracket
else if (it->tokenType == Token::TokenType::Right_Paran)
{
    while (it->tokenType != Token::TokenType::Left_Paran)
    {
        //pop operators from the stack onto the output queue
        if (!shuntingYardStack.top().tokenType != Token::TokenType::Left_Paran &&
        {
            rpnList.push_back(shuntingYardStack.top());
            shuntingYardStack.pop();
            break;
        }
    }
    shuntingYardStack.pop();
}
```

4.1.3 RPN Solver

The RPN solver is given a list of tokens in postfix notation, as a result of the shunting yard. This list is parsed from left to right until only one number is left in the list, this is the answer and is displayed to the user.

RPN Solver Algorithm (Structured English)

While there are tokens to read:
    If it's an operator:
        Pop the top 2 numbers from the stack
        Switch to the correct operator and do the calculation
    Else if it's a number:
        Push it to the number stack

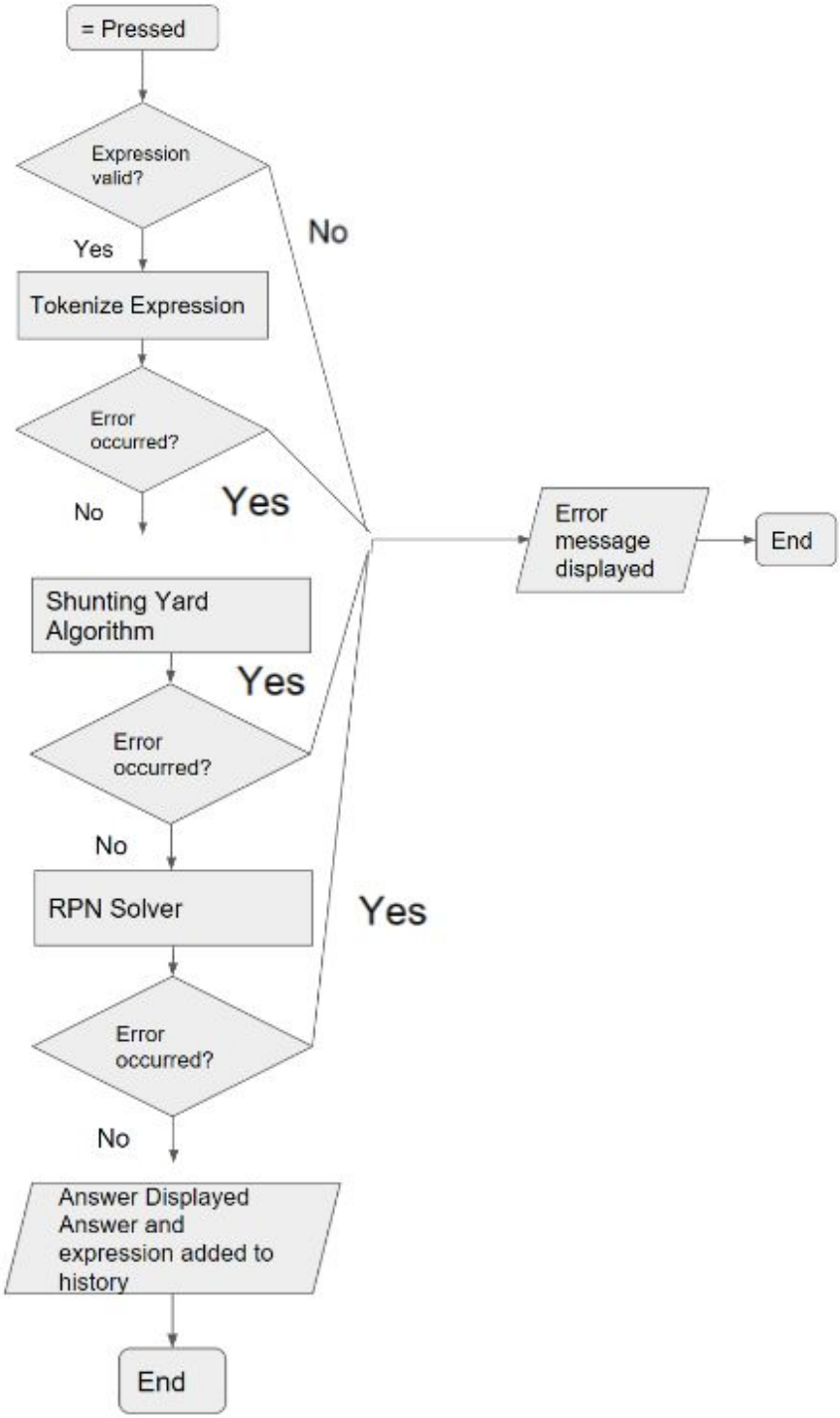
Example Code

```
20 double evalRPN(std::list<Token> & rpnList) {
21     std::stack<double> nums;
22     for (std::list<Token>::iterator it = rpnList.begin(); it != rpnList.end(); ++it) {
23         if (it->tokenType == Token::TokenType::Operator) {
24             // pop two numbers off the stack
25             double a = nums.top();
26             nums.pop();
27             double b = nums.top();
28             nums.pop();
29             // evaluate and push the result back
30             switch (it->symbol)
31             {
32             case '+': nums.push(a + b); break;
33             case '-': nums.push(b - a); break;
34             case '*': nums.push(b * a); break;
35             case '/': nums.push(b / a); break;
36             case '^': nums.push(pow(b, a)); break;
37             default:;
38             }
39         }
40         else if (it->tokenType == Token::TokenType::Number) {
41             // push a number into the stack
42             double n = it->value;
43             std::cout << "n =" << n << std::endl;
44             nums.push(n);
45         }
46         else
47             std::cout << it->tokenType << " error" << std::endl;
48     }
```

4.1.4 Calculation Button Press

Pressing the "=" button calls the functions mentioned in the previous paragraphs in this order, tokenization, shunting yard then finally the RPN solver as seen below in the flow chart.

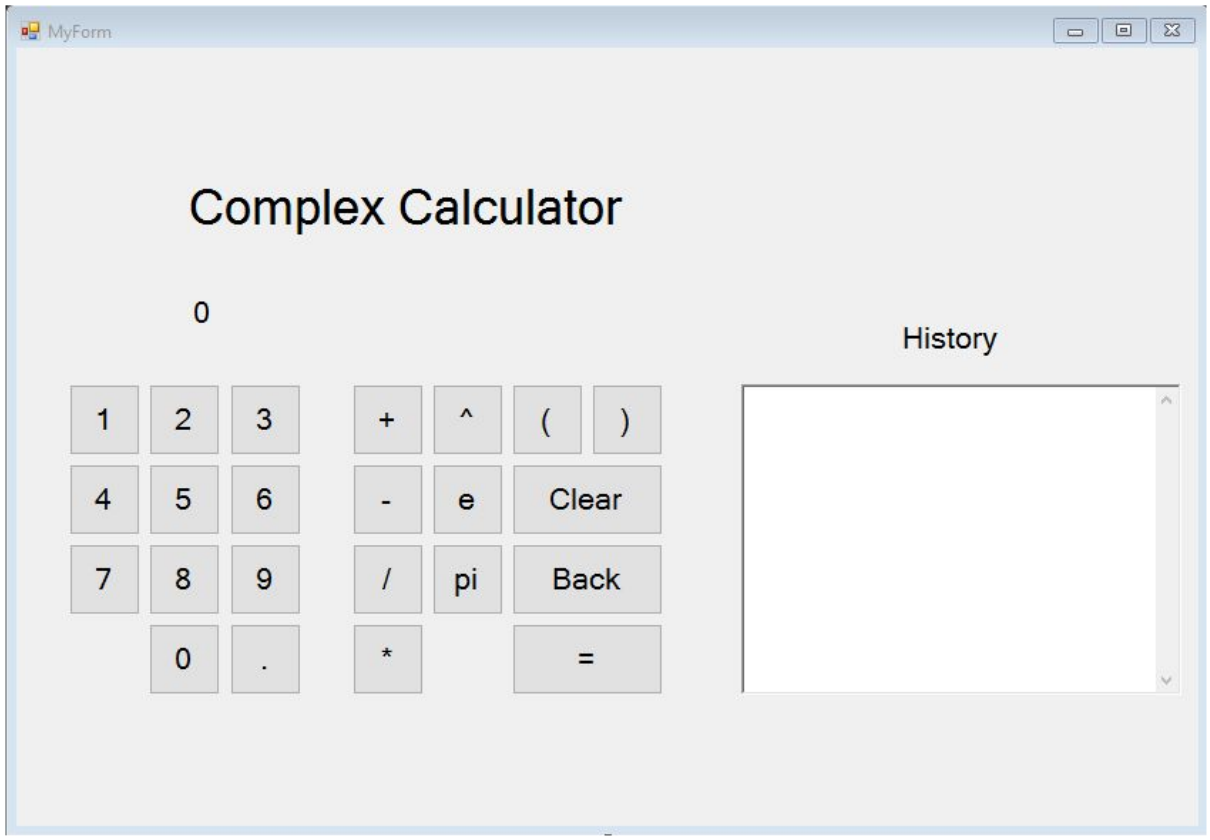
Flow Chart



4.1.5 Graphical User Interface (GUI)

The GUI of any type of software needs to be clear and coherent, the user will need to quickly understand what the GUI is trying to get the user to understand otherwise they will get bored or confused and won't use that piece of software again due to the use not being very clear. This is why the GUI is designed to look the same as most other physical and digital calculators, due to affordance this means that the user will probably have seen this type of calculator before as well as the symbols used on all calculators are universal meaning learning how to use this calculator will be easy.

GUI Image



4.2.Development

4.3. Version-1

The objective of Version 1 was to create a calculator capable of simple calculations using the tokenization method, shunting yard algorithm(1) and a RPN solver. The expressions expected to work at this version was “2+2”, “5/2” etc. As well as accepting floating point numbers e.g 5.5, 2.5, 5.3. This level was mostly just getting the core functions down to make improving the calculator much easier.

Test	Use Case	Expected Outcome	Actual Outcome	Result	Fixed?
1	= pressed without any input.	Nothing happens or user notified	Calculator gave a result 0	Failed	No
2	User enters 1+1	Calculator gives a result 2	Calculator gave a result 2	Passed	N/A
3	User enters 5/2	Calculator gives a result 2.5	Calculator gave a result 2.5	Passed	N/A
4	User enters -1	Nothing happens or user notified	Crashed	Failed	No
5	User enters 1++1	Error occurred, user notified	Crashed	Failed	No
6	User enters 1.5+2.5	Error occurred, user notified	Crashed	Failed	Yes
7	User enters 2*2	Calculator gives a result 4	Calculator gave a result 4	Passed	N/A
8	User enters 5-2	Calculator gives a result 3	Calculator gave a result 3	Passed	N/A
9	User enters 2^2	Calculator gives a result 4	Calculator gave a result 4	Passed	N/A
10	User pressed a button	Calculator displays number on screen	Calculator displayed number on screen	Passed	N/A

4.4. Version-2

Version 2 was about extending the calculator to be able to solve expressions such as 2+3*2 and implementing brackets so more complicated expressions can be inputted such as ((2+3)-(4-2)*2). Also implementing a Clear button to clear the expression to start over.

Test	Use Case	Expected Outcome	Actual Outcome	Result	Fixed?
1	User enters (2*2)	Calculator gives a result 4	Calculator gave a result 4	Passed	N/A
2	User enters (2+2)*(5/2)	Calculator gives a result 10	Calculator gave a result 10	Passed	N/A
3	User enters ((2+3)-(4-2)*2)	Calculator gives a result 4	Calculator gave a result 4	Passed	N/A
4	User enters (2*(5/2)+(9-4.3))	Calculator gives a result 9.7	Calculator gave a result 9.7	Passed	N/A

5	User enters ((5^3)*(3+2)/(2^2))*2.34	Calculator gives a result 365.625	Calculator gave a result 365.625	Passed	N/A
6	User presses “Clear” button	Expression is cleared	Expression cleared	Passed	N/A

4.5. Version-3

Version 3 was about extending the calculator to manage all types of numbers, implementing constants like pi and e, writing in error messages to stop the program from crashing if an error occurred due to the user's input as well as a back button.

Test	Use Case	Expected Outcome	Actual Outcome	Result	Fixed?
1	User enters (20.435+32.29032)/2	Calculator gives a result 26.3627	Calculator gives a result 26.3627	Passed	N/A
2	User enters e^4	Calculator gives a result 54.5982	Calculator gave a result 54.5982	Passed	N/A
3	User enters (e^3.24)*(4/2.3)	Calculator gives a result 44.4065	Calculator gave a result 44.4065	Passed	N/A
4	User enters 3..2	Error message appears on console	Error message appeared on console	Passed	N/A
5	User enters 2*pi	Calculator give a result 6.28319	Calculator gave a result 6.28319	Passed	N/A
6	User enters 35+45, then presses back button	Calculator displays 35+4	Calculator displayed 35+4	Passed	N/A

5. Release/Acceptance Tests/Deployment

The testing of the calculator was very successful as the majority of tests were passed, however there was suppose to be another version of the calculator but it was never implemented due to time restraints. If there was more time to complete and ship the calculator there would be more operations like sine, cosine, ln, log, etc. Due to needing to tweak the tokenizer to recognize the symbols, even though this wouldn’t take much time I would have to change the shunting yard method to take the functions into account. Considering this extra programming, there just simply wasn't enough time to achieve this.

However, the testing for the versions of the calculator that were completed was successful due to the investment of a good core foundation of functions to solve the expressions (tokenization, shunting yard and the RPN solver). Each function set up the ease of adding more operations and gives a correct answer to up to 5 decimal places.

The calculator was a success overall a success and due to it being a piece of software, it can be updated by the developer or even modified by a user if so desired on demand on most machines.

References

Shunting Yard Algorithm, Wikipedia (Accessed 5th March 2017)
https://en.wikipedia.org/wiki/Shunting-yard_algorithm

Shunting Yard Algorithm, Brilliant.org (Accessed 6th March 2017)
https://brilliant.org/wiki/shunting-yard-algorithm/

Lexical Analysis, Wikipedia (Accessed 5th March 2017)
https://en.wikipedia.org/wiki/Lexical_analysis

Appendix-A: Software Requirement Specification

Appendix-B: MoSCoW Analysis

Appendix-C: Scrum Board