EIE3109 Lab01 Report

Author: LIU Minghao

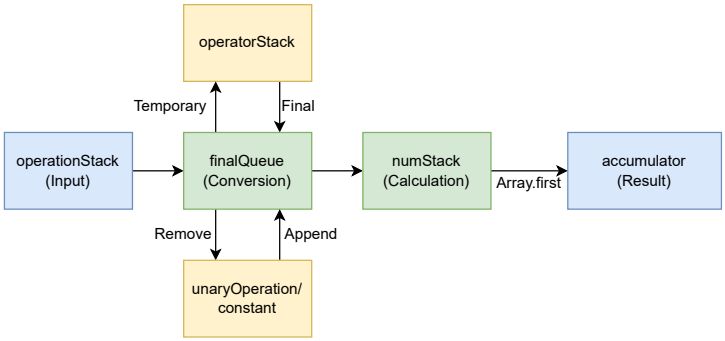
1. Sequential processing of calculator operation

- As a calculator, it is important to implement these four operations. In the "Lab 1 (instructions)" given by the instructor, the `performPendingBinaryOperation()` method is used to handle binary operations. This method works by recording the last operation, and the last value as a **pendingBinaryOperation**. For the next value entered, the `performPendingBinaryOperation()` method is called to perform the operation and the result is stored in the **accumulator** and the next operation uses the **accumulator** as the first value;
- However, pendingBinaryOperation does not lend itself to a complex quadratic operation. For example, if the user enters "1+2x3" and presses equals, the logic of the code becomes 1+2=3, and then 3x3 is used to get the result 9;
- I'm borrowing from my previous experience in **COMP2011 Data Structure** class, using the **Reverse Polish Notation (RPN)** to handle the computational logic, which is a method used in computers to find values, and the process uses **Stack** to store them;
- The advantage of the **RPN** algorithm is that it can handle complex expressions with clear and understandable logic. This allows the calculator to satisfy the logic of **multiplying and dividing before adding and subtracting** in complex quadratic operations;
- The example:

Infix expression	Postfix expression
1+2	12+
1+2x3	123x+
1+2x3-4	123x+4-
1+2x3-4/5	123x+45/-

Flow Chart

- The flow chart of the **RPN** algorithm is as follows:



- As shown in the above flow chart, the **RPN** algorithm is divided into two parts, the first part is to convert the **infix expression** into a **postfix expression**, and the second part is to calculate the result of the **postfix expression**;

- The operatorStack is to temporarily store the operators in the **postfix expression**, and will append back to the finalQueue before calculation;
- The unaryOperation/constant is to calculate the result of the unary operator/constant and push it into the **finalQueue**;

Data Structure

- Throughout the computation process, **Stack** is utilized to store all operators and values;
- I use `operationStack` as a **stack** to store all operators and values in **infix expressions**, this stack is essentially an **array of string** types;

```
private var operationStack: [String] = []
func operationPressed(button: CalcuButton)
    operationStack.append(self.value)
    if button != .clear && button != .equal { operationStack.append(button.rawValue) }
```

- As shown in the above code, I have stored **self.value** and **button.rawValue** on the **operationStack** in the `operationPressed()` method;

```
var finalQueue: [String] = []
var operatorStack: [String] = []
var numStack: [Double] = []
```

- finalQueue is to store the **postfix expression** converted from the **infix expression**, which is a string array;
- operatorStack is to temporarily store the operators in the **postfix expression**, which is a string array;
- numStack is to store the values in the **postfix expression**, which is a double array;

```
let operators: [String: (precedence: Int, isLeftAssociative: Bool)] = [
    "+": (precedence: 1, isLeftAssociative: true),
    "-": (precedence: 1, isLeftAssociative: true),
    "x": (precedence: 2, isLeftAssociative: true),
    "/": (precedence: 2, isLeftAssociative: true)
]
```

- The above code is a **dictionary** that stores the **precedence** and **associativity** of the operator, which is used to determine the order of the operator in the **postfix expression**;

Logic Workflow

- The logic of the **RPN** algorithm is to convert the **infix expression** into a **postfix expression** and then calculate the result;

```
for op in operationStack
    // binary operator
    if let operatorData = operators[op] {
        let currentOperatorPrecedence = operatorData.precedence
        let currentOperatorIsLeftAssociative = operatorData.isLeftAssociative
        while let topOperator = operatorStack.last,
        let topOperatorPrecedence = operators[topOperator]?.precedence,
        (currentOperatorIsLeftAssociative && currentOperatorPrecedence <= topOperatorPrecedence) ||
        (!currentOperatorIsLeftAssociative && currentOperatorPrecedence < topOperatorPrecedence) {
        finalQueue.append(operatorStack.removeLast())
        }
        operatorStack.append(op)
    }
```

- If the **precedence** and **associativity** of the operator in the **infix expression** are lower than the **precedence** and **associativity** of the operator in the **postfix expression**, the operator in the **postfix expression** is popped out and added to the **finalQueue**. If the **precedence** and **associativity** of the operator in the **infix expression** are higher than the **precedence** and **associativity** of the operator in the **postfix expression**, the operator in the **infix expression** is pushed into the **operatorStack**;

```
for op in operationStack
    // number, constant, unary operator
    if let number = Double(op) {
        finalQueue.append(String(number))
    } else if op == "e" {
        finalQueue.removeLast()
    } else if op == "π" {
        finalQueue.removeLast()
    } else if op == "sin" {
        finalQueue.removeLast()
    } // cos, %, +/- is the same as sin
```

- The above code is also in the same for loop of the binary operator;
- For unary operators, **finalQueue** stores the value which has computed by the unary operator (sin, cos, +/-, %), and the unary operator and the original values will not be appended in the **finalQueue**;
- For the number in operationStack, it is pushed into the **finalQueue**;
- For the constant, it will become the double type and pushed into the **finalQueue**;

```
while !operatorStack.isEmpty { finalQueue.append(operatorStack.removeLast()) }
```

- After the **infix expression** is converted into a **postfix expression**, the remaining operators in the **operatorStack** are popped out and added to the **finalQueue**;

```
if let number = Double(op) { numStack.append(number) }
else { let operand2 = numStack.removeLast()
    let operand1 = numStack.removeLast()
    switch op
    case "+": // -, x, / is the same as +
        numStack.append(operand1 + operand2)
}
```

- The above code is in for loop to calculate the result of the **postfix expression**, which is to pop out the last two values in the **numStack** and perform the corresponding operation according to the operator in the **finalQueue**. The result is pushed into the **numStack**;

```
accumulator = ((numStack.first ?? 0)*100000).rounded()/100000
```

- The above code is to find the result of the **postfix expression** and round it to 5 decimal places;

```
func resetCalculator()
    self.value = "0"
    isUserEnteringNumber = false
    operationStack = []
    accumulator = 0
```

- The above code is used to reset the calculator, which is called in the `operationPressed()` method when the user presses the **clear** button or the next operation after the **equal** button;

2. Advanced Functions

History of the input

- In this part, I have implemented the the **input history** function;

```
private var valueListArray: [String] = []
```

- Use the **valueListArray** to store the user input button history;
- The **valueListArray** is a string array, and the elements in the array are the **value** and **calculator**;

```
func valueListPressed(button: CalcuButton) {
    valueListArray.append(button.rawValue)
    var index: Int = 0
    var cache: String = ""
    // the printing rules for lists
}
```

- The index is used as a pointer to the **valueListArray**;
- The cache is used to store the string to be printed;
- And also there will be some rules of the printing, such as push "sin' button, it will print out "sin(self.value)"

```swift
func findLastNonDigitIndex(stringArray: [String]) -> Int {
    return index
}

// to find the last operator Index
func findLastOperatorIndex(stringArray: [String]) -> Int {
    return index
}
```

- The above two methods are used to find the last non-digit and operator index in the **valueListArray**, and the index can use this two methods to find the value;
- As use the advanced functions, user can view the history of the input, and the history will be printed in the **valueList**;

Omit numerical calculations

- As my sequential processing of calculator operation, the system support the omit numerical calculations;
- For example, the user can input "3+-x" and the equal button, the logic will be "3+3-3x3", and the result is "-6";
- Another example, the user can input "3x/2" and the equal button, the logic will be "3x3/2", and the result is "4.5";
- The logic of the function is when the last operator is the final input, and user not input the value, the system will use the **self.value** (the vlaue show in screen) as the value;
- And there will be some examples:

Input	Calculation	Result
1+3x-sin()	1+3x3-sin(3)	9.94766
1+πsin()x-	1+sin(π)xsin(π)-sin(π)	0.9482
1+sin()x-	1+sin(1)xsin(1)-sin(1)	0.98285