Evaluate Mathematical Expressions Assignment

In this assignment we are going to implement mathematical expression evaluations. Given a mathematical expression like 3 + 4*6 or 3 *(3+2) – 5, the program should return 27 or 10, respectively.

We are going to use the Shunting-Yard algorithm for converting the infix expression to postfix (Reverse Polish Notation), and a stack-based evaluation method to compute the result.

To do that, we need to define two stacks stk1 and stk2. The Standard Template Library (STL) in C++ has a built-in library for stack. To use we need to include the stack library as: #include <stack>.

```
#include <iostream>
#include <stack>
#include <cmath>
using namespace std;
```

Let us prepare for the declaration of these stacks. The first thing we need to do is to declare an enum that defines the element type as follows:

enum ElementType { NUM, OP};

Then, we define the stack elements as class Element:

```
enum ElementType { NUM, OP};
class Element {
    private:
        string v;
        ElementType type;
    public:
        string getString() { return v; }
        ElementType getType() { return type; }
        Element(string s, ElementType t) {
            v = s;
            type = t;
        }
        Element() {
            v = "";
            type = NUM;
        }
};
```

The above class stores two variables: a string v and ElementType type. The variable v can be either a number string or an operator string like “+”, “-“, “*” … etc. the variable type determines if v stores a number string (type=NUM) or an operator string (type = OP).

Now, define the stack as:

```
stack<Element> stk1, stk2;
```

We are going to consider the operators: +, -, *, /, ^ (for power), (,) only.

The precedence function is defined by the function getPrecedence as follows:

```
int getPrecedence(string op) {
    if (op == "^") return 3;
    if (op == "*" || op == "/") return 2;
    if (op == "+" || op == "-") return 1;
    return 0;
}
```

For debugging purposes, we are going to define a recursive function printStack to display the content of the stack; this function will pop all elements display them on the screen and push them back into stack again.

```
void printStack(stack<Element> s) {
    if (s.empty()) return;

    Element t = s.top();
    s.pop();

    printStack(s);

    cout << t.getString() << " ";
    s.push(t);
}
```

The next function is a Boolean function that returns true when the string is an operator and false otherwise:

```
bool isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
}
```

Then comes the function postfix that convert the infix expression into postfix notation:

```
void postifx(string s) {
    char oper;
    string temp;
    Element t("$", OP);
    stk2.push(t);
```

Note: the function postfix takes on parameter string s, that is the expression provided by the user like 3+2*4.

Now we loop to the end of the expression for several cases (big for loop):

```
for (int i=0; i<s.length(); i++) {
        … // several cases
}
```

The cases are:

1) Ignore space:
```
if (s[i] == ' ') continue; // ignore spaces
```
2) If a left parenthesis character is encountered, the push on the stack stk2:
```
else if (s[i] == '(') { // Left parenthesis
    stk2.push(Element("(", OP));
    continue;
}
```
3) If a right parenthesis character is encountered, then you need to move elements from stk2 to stk1 until we encounter in the stk2 the left parenthesis character. If left parenthesis character is not found and we reach the bottom of the stack then an error message is displayed (mismatch parenthesis).

```
else if (s[i] == ')') { // right parenthsis:
    // move elements from stk2 to stk1 until left parenthsis found.
    while (!stk2.empty()) {
        Element t = stk2.top();
        if (t.getString() == "(") { // stop when left parenthsis found
            stk2.pop();
            break;
        } // if
        if (t.getString() == "$") { // if reach bottom of stack, then error
            throw string("Error... Mismatch Parenehtis");
        } // if
        // move elements from st2 to stk1
        Element k = stk2.top();
        stk2.pop();
        stk1.push(k);
    } // while
} // if (s[i] == ')')
```

4) If a digit is encountered, then collect all the digits in string t, and store them into stack stk1.

```
// read number:
else if (s[i]>='0' && s[i] <= '9') { // digit found
    // collect all digits into string t
    string t = "";
    while (s[i]>='0' && s[i] <= '9') {
        t += s[i];
        i++;
    } // while
    i--;
    stk1.push(Element(t, NUM)); // push string t to stk1

} // if
```

5) If an operator is encountered, then push it into stack stk2 only when the precedence if the operator is greater than the operator at the top of stk2. Otherwise, move the top operators from stk2 to stk1.

```
else if (isOperator(s[i])) {  // Operator:
    string oper(1, s[i]); // convert incoming operator char to string
    int k = getPrecedence(oper); // get its precedence
    while (!stk2.empty()) {
        Element t = stk2.top(); // get top element in stk2
        int j = getPrecedence(t.getString()); // get its precedence
        if (k<=j) { // if incoming prcedence <= top precedence then
            // move operator from stk2 to stk1
            stk1.push(t);
            stk2.pop();
        } // if
        else { // otherwise, incoming precendence > top precedence then
            Element t(oper, OP); // push incoming operator to stk2
            stk2.push(t);
            break;
        } // else
    } // while
} // if isOperator
else throw s[i] + string(" Invalid character encountered in expression " );
```

At the end of this big 'for' loop that includes all the above cases. We should empty what is left over in stk2 and move it into stk1.

```
while (!stk2.empty()) {
    Element t = stk2.top();
    if (t.getString() == "$") break;
    stk2.pop();
    stk1.push(t);
}
```

After completing the postfix function, we should get the postfix notation of the infix expression typed by the user. Now we are going to the evaluation function that evaluates the postfix expression generated and produces the result. At the start of the evaluate function to empty stk1 into an array of elements s. Here is the evaluate function:

```
int evaluate() {
    int size = stk1.size();
    Element s[size];
    int index = size-1;
    // Empty stk1 into a string s
    while (!stk1.empty()) {
        Element t = stk1.top();
        stk1.pop();
        s[index] = t;
        index--;
    }
    // s contains the postfix notation
```

After getting the postfix expression in variable array s, we need to evaluate it. The evaluation step is easy, start a loop from 0 to the end of the array s; once you find an operator, then you can compute the last two numbers, if no numbers found then error message should be display like “Invalid expression”. The start of this step is as follows:

```
// s contains the postfix notation
for (int i=0; i<size; i++) {
    if (s[i].getType() == OP) { // if operator
        if (i<2) throw string("Invalid Expression ");
        if (s[i-1].getType() != NUM) throw string("Invalid Expression");
        if (s[i-2].getType() != NUM) throw string("Invalid Expression");
        int a = stoi(s[i-1].getString()); // get previous index number
        int b = stoi(s[i-2].getString()); // get previous previous index number
        int r; // for storing result
        string op = s[i].getString(); // get the operator
        if (op=="+") r = a+b;
        else if (op=="-") r = b-a;
        else if (op=="*") r = a*b;
        else if (op=="/") {
            if (a == 0) throw string("Division by zero\n");
            r = b/a;
        }
        else if (op=="^") r = pow(a, b);
        s[i-2] = Element(to_string(r), NUM);
        // shif the array twice from i-2.
        for (int j=i+1; j<size; j++) {
            s[j-2] = s[j];
        }
        size -= 2; // adjust size
        i -= 2; // adjust i
    }
}
return stoi(s[0].getString()); // return final result
```

Note: error checking is done via the throw command in C++, that throws string expressions as an error signal. The errors occur when you have an operator without two numbers in the previous two indices. Another error occurs when you divide by zero.

The main function:

```cpp
int main() {
    string s;

    cout << "Enter your expression: ";
    getline(cin, s);


    int v = 0;
    try {
        postifx(s); // call postfix function
        v = evaluate(); // call evaluate
        cout << "v=" << v << endl;
    }
    catch (string s) {
        cout << s << endl;
    }

    return 0;
}
```

Note: the main catches the error that is thrown from function evaluate.

Test cases:

1) Try to input the expression: 5+2*4, the result should be 13
2) Try to input the expression: 3*(2+4*(5-2)-5), result should be: 27
3) Try to input the expression: 3*+2, result: Invalid expression
4) Try to input the expression: 3/(4-4), result: Division by zero.
5) Try to input the expression: 3+4&4, result: & invalid character encountered in expression.

Now let us implement the code in Python:

Define two stacks stk1 and stk2 as lists:

```python
stk1 = []
stk2 = []
```

Define operators' precedence as a dictionary (associative array):

```python
p = {'$': 0, '(': 1, '+': 2, '-': 2, '*': 3, '/': 3, '^': 4}
```

Get the input mathematical expression from the user:

```python
s = input("Enter your expression: ")
```

We need a function to check if a character is an operation "+", "-", "*", "\", "^", "(":

```python
def isOperator(s):
    return s in ["+", "-", "*", "/", "^"]
```

Then we write the postfix function, that takes the mathematical expression string. The function will prepare for the main while loop to convert the input infix to postfix. Specifically, it initializes stk1 to empty and stk2 to include a dollar string at the bottom of the stack. The function defines the control variable k=0 to loop over the input string, then the while loop starts.

```
def postfix(s):
    global stk1, stk2
    stk1 = []
    stk2 = ['$']
    s = list(s)
    k = 0
    while k < len(s):
```

There are several cases inside the while loop:

Case 1: spaces encountered:

```
if s[k] == " ": k+=1; continue
```

Case 2: a right parenthesis encountered:

```
if s[k] == "(":
    stk2.append(s[k])
```

Case 3: a left parenthesis encountered:

```
elif s[k] == ")":
    while True:
        if stk2[-1] == "$":
            raise ValueError("invalid expression - Mismatch Parenthsis")
        c = stk2.pop()
        if c == "(": break
        stk1.append(c)
```

A digit encountered:

```
elif s[k] >= "0" and s[k] <= "9":
    t = ""
    while k < len(s) and s[k]>="0" and s[k] <= "9":
        t += s[k]
        k += 1
    k -= 1
    stk1 += [int(t)]
```

An operator is encountered:

```
elif isOperator(s[k]):
    c = stk2.pop()
    while p[s[k]] <= p[c]:
        stk1.append(c)
        c = stk2.pop()
    stk2.append(c)
    stk2.append(s[k])
```

The last case is an invalid character that raises an error:

```
else: raise ValueError(s[k] + " Invalid Character found ")
```

Finally, we need to increment k to the next element in the input string.

```
k += 1
```

This will end the while loop. After that we need to loop through stk2 to move the elements from stk2 to stk1:

```
while stk2:
    c = stk2.pop()
    if c == '$': break
    stk1.append(c)
print('postfix: ', stk1)
```

It is time to write the evaluate function, the first thing is we are going to write an evaluate operation 'evalOp' that takes the stk1 and the index of the operator and evaluate it:

```
def evalOp(stk1, i):
    if i < 2: raise ValueError("Invalid expression - can not compute.")
    k = stk1[i-1]
    j = stk1[i-2]
    if type(k) != int or type(j) != int: raise ValueError("Invalid expression...")
    if stk1[i] == "+": r = k + j
    elif stk1[i] == "-": r = j - k
    elif stk1[i] == "*": r = k * j
    elif stk1[i] == "/":
        if k ==0: raise ValueError("Divvision by zero")
        else: r = j/k
    elif stk1[i] == "&": r = j**k
    return r
```

Then the evaluate function that loops in stk1 to compute the postfix notation:

```
def evaluate():
    i = 0
    size = len(stk1)
    while i < size:
        if isOperator(stk1[i]):
            r = evalOp(stk1, i)
            stk1[i-2] = r
            for k in range(i+1, len(stk1)):
                stk1[k-2] = stk1[k]
            i -= 2
            size -= 2
        i += 1
    return stk1[0]
```

Finally, we have the main program that calls postfix and evaluate:

```
try:
    postfix(s)
    r = evaluate()
    print('result=', r)
except ValueError as v:
    print(v)
```

Test cases:

1) Try to input the expression: 5+2*4, the result should be 13
2) Try to input the expression: 3*(2+4*(5-2)-5), result should be: 27
3) Try to input the expression: 3*+2, result: Invalid expression – can not compute
4) Try to input the expression: 3/(4-4), result: Division by zero.
5) Try to input the expression: 3+4&4, result: & invalid character found.

What to submit:

- Submit the correct programs that work on all the above test cases to canvas.
- Implement a Java program that works on all the above test cases. Upload it to canvas.