

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)

Институт информационных технологий, математики и механики

ЛАБОРАТОРНАЯ РАБОТА
на тему:
**«ВЫЧИСЛЕНИЕ АРИФМЕТИЧЕСКИХ
ВЫРАЖЕНИЙ (СТЕКИ)»**

Выполнил(а): студент группы 3822Б1ФИ1
_____/ Петров Н.Д. /
Подпись

Проверил: к.т.н., доцент каф. ВВиСП
_____/ Кустикова В.Д. /
Подпись

Нижний Новгород
2023

Оглавление

Введение	3
1 Постановка задачи.....	4
2 Руководство пользователя.....	5
2.1 Приложение для демонстрации работы стека.....	5
2.2 Приложение для демонстрации работы парсера	5
3 Руководство программиста	7
3.1 Используемые алгоритмы	7
3.1.1 Стек.....	7
3.1.2 Постфиксная форма	8
3.2 Описание классов.....	10
3.2.1 Класс Stack.....	10
3.2.2 Класс Parser.....	12
Заключение	14
Литература	15
Приложения	16
Приложение А. Реализация класса Stack.....	16
Приложение Б. Реализация класса Parser	17
Приложение В. Реализация класса sample_parser.....	20

Введение

Запись арифметических выражений в постфиксной форме является одним из методов представления математических выражений. В этой форме операнды располагаются перед операторами, что устраняет неоднозначность порядка операций. При реализации алгоритма преобразования инфиксной записи в постфиксную часто используется стек.

Стек – это абстрактная структура данных, основанная на принципе "последним пришёл, первым вышел" (Last In, First Out, LIFO). Это означает, что последний элемент, добавленный в стек, будет извлечен первым.

1 Постановка задачи

Цель – реализация классов **stack** и **parser**, предназначенных для преобразования инфиксной записи арифметического выражения в постфиксную.

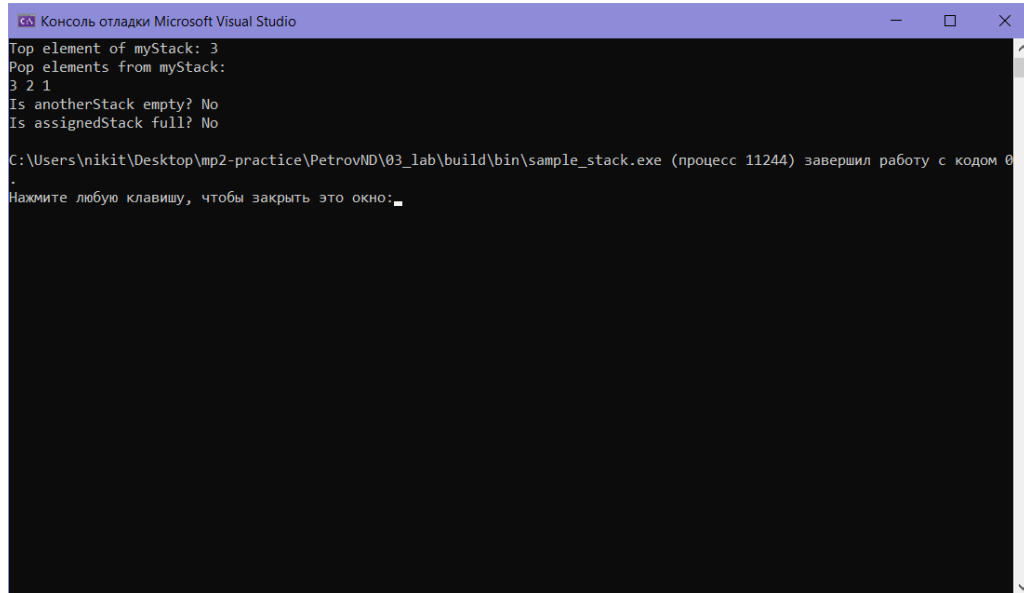
Задачи:

1. Исследовать тематическую литературу.
2. Реализовать класс **stack**.
3. Реализовать класс **parser**.
4. Провести тестирование разработанных классов для проверки их корректной работы.

2 Руководство пользователя

2.1 Приложение для демонстрации работы стека

1. Запустить sample_stack.exe. В результате появится следующее окно (рис. 1).



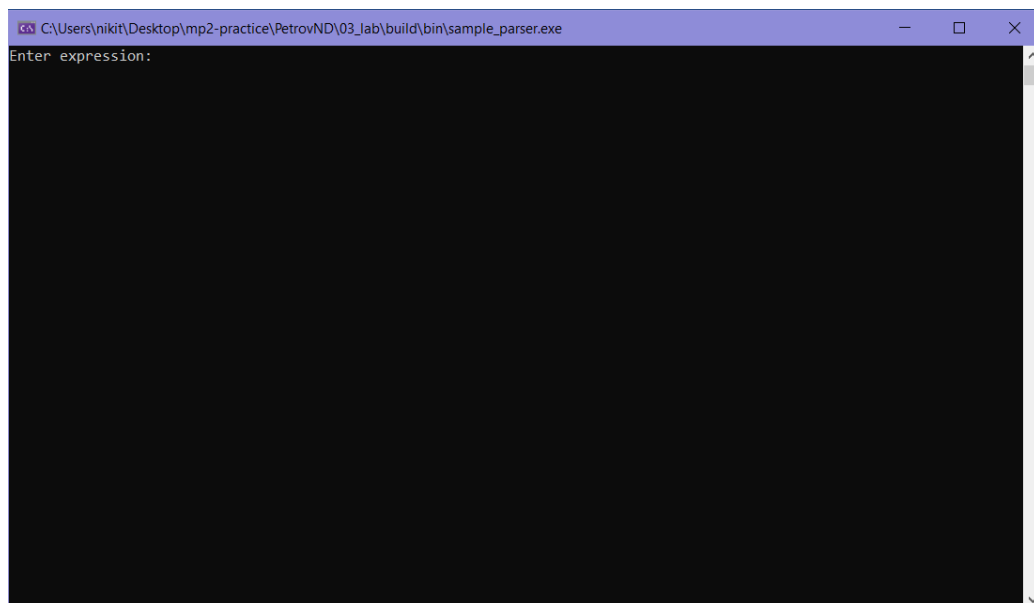
```
Консоль отладки Microsoft Visual Studio
Top element of myStack: 3
Pop elements from myStack:
3 2 1
Is anotherStack empty? No
Is assignedStack full? No

C:\Users\nikit\Desktop\mp2-practice\PetrovND\03_lab\build\bin\sample_stack.exe (процесс 11244) завершил работу с кодом 0
Нажмите любую клавишу, чтобы закрыть это окно: _
```

Рис. 1. Основное окно приложения sample_stack

2.2 Приложение для демонстрации работы парсера

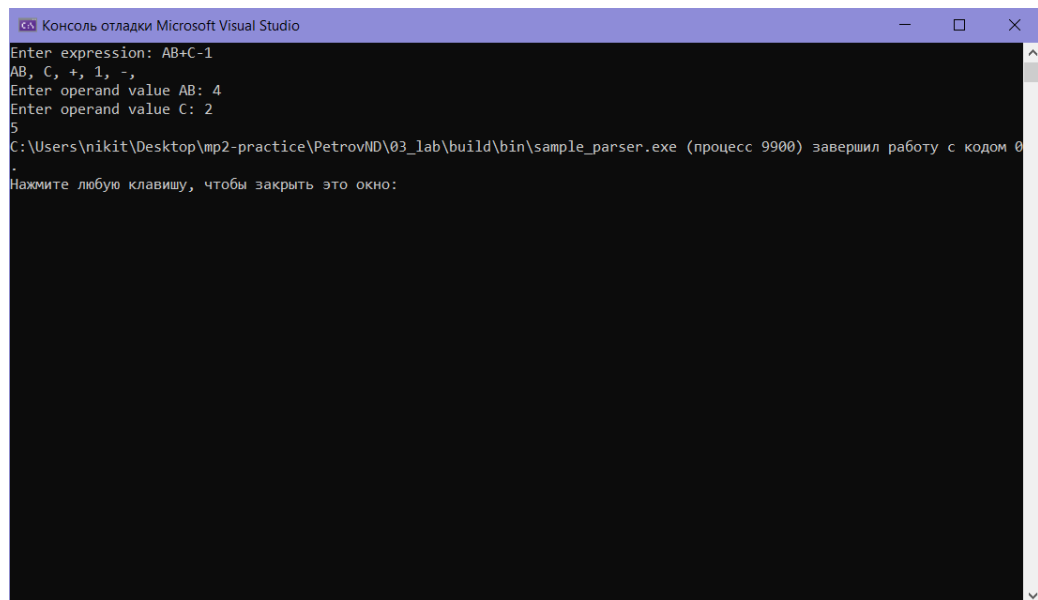
1. Запустить sample_parser.exe. В результате появится следующее окно (рис. 2).



```
C:\Users\nikit\Desktop\mp2-practice\PetrovND\03_lab\build\bin\sample_parser.exe
Enter expression:
```

Рис. 2. Основное окно приложения sample_parser

2. Затем необходимо ввести арифметическое. После ввести значения операндов (рис. 3).



```
Консоль отладки Microsoft Visual Studio
Enter expression: AB+C-1
AB, C, +, 1, -,
Enter operand value AB: 4
Enter operand value C: 2
5
C:\Users\nikit\Desktop\mp2-practice\PetrovND\03_lab\build\bin\sample_parser.exe (процесс 9900) завершил работу с кодом 0
.
Нажмите любую клавишу, чтобы закрыть это окно:
```

Рис. 3. Основное окно приложения sample_tset

3 Руководство программиста

3.1 Используемые алгоритмы

3.1.1 Стек

Стек – это абстрактная структура данных, основанная на принципе "последним пришёл, первым вышел" (Last In, First Out, LIFO). Поддерживает операции просмотра значения верхнего элемента, удаление верхнего элемента, добавление элемента на вершину, проверка на полноту и пустоту.

Пример стека размера 4 (рис. 4):

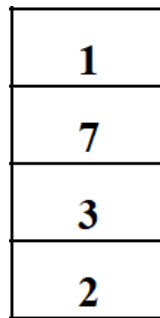


Рис. 4. Представление стека

Операции над стеком:

1. Просмотр верхнего элемента, возвращает его значение (рис. 5).

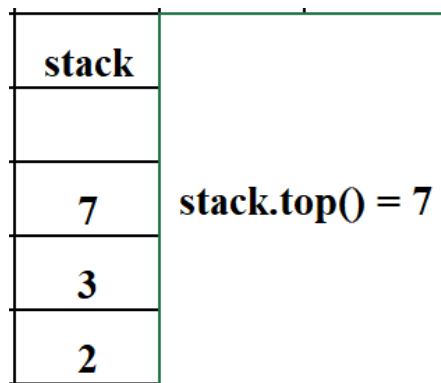


Рис. 5. Функция top()

2. Добавление элемента. Если стек не полон, добавляется элемент на позицию `top_+1`, при этом `top_` увеличивается на 1 (рис. 6).

top_=2	top_=3
stack	stack.push(1)
	1
7	7
3	3
2	2

Рис. 6. Функция push()

3. Удаление верхнего элемента. Осуществляется путём уменьшения на единицу индекса верхнего элемента стека (**top_**).
4. Проверка на пустоту. Возвращает **true** если **top_ = -1**.
5. Проверка на полноту. Возвращает **true** если стек полон (**top_ = maxSize-1**).

3.1.2 Постфиксная форма

В постфиксной записи каждое арифметическое выражение записывается так, чтобы каждый оператор следовал после своих операндов. Например, обычное инфиксное выражение "2 + 3" в постфиксной записи будет выглядеть как "2 3 +".

Преимущества постфиксной записи включают удобство для компьютерной обработки и вычислений, а также отсутствие необходимости использования скобок для указания порядка операций.

Алгоритм преобразования постфиксной записи

1. Для операций вводится приоритет: операция умножение деление 3, сложение вычитание 2, левая скобка 1, присваивания 0. Для правой нет (просто сигнал для перекладывания элементов).
2. Для хранения используются 2 стека: первый для операндов и результата, второй для операций.
3. Исходное выражение просматривается с лева на право:
 - 3.1 Операнды по мере их появления помещаются в стек 1
 - 3.2 Символы операции, левой скобки помещаются в стек 2. Причём выполняется следующее правило: если операция имеет приоритет более низкий чем операция на верхушке стека 2, то все операции с большим или равным приоритетом перекладывается в стек 1. Сама текущая операция кладётся в стек 2. Если пришла

левая скобка, то она просто кладётся в стек 2. Если пришла правая скобка, то изымаются элементы из стека 2 и переносятся в стек 1 до тех пор, пока не будет изъята левая скобка из стека 2 или он не будет пуст, левая скобка просто удаляется.

Пример:

Инфиксное выражение: $(a+b*c)*(c/d-e)$

Постфиксная запись: $abc*+cd/e-*$

- Стек постфиксной формы:

												*
												-
											e	e
										/	/	/
									d	d	d	d
						c	c	c	c	c	c	c
					+	+	+	+	+	+	+	+
					*	*	*	*	*	*	*	*
				c	c	c	c	c	c	c	c	c
		b	b	b	b	b	b	b	b	b	b	b
a	a	a	a	a	a	a	a	a	a	a	a	a

- Стек операторов:

				*	*				/	/	-	-	
		+	+	+	+		((((((
((((((*	*	*	*	*	*	*	

Алгоритм вычисления. На входе строка в постфиксной. Форме. Выход – число.

1. Пока не достигнут конец последовательности читаем лексему.
 - 1.1 Если прочитан операнд кладём его значение в стек
 - 1.2 Если прочитан знак операции - то изымаем из стека 2 операнда, выполняем операцию (в обр. порядке), кладём результат в стек. (стек значений).
2. Если достигнут конец арифметического выражения – то завершаем работу. Если на входе было корректное выражение, то в рабочем стеке будет результат.

Пример:

- 1) *Инфиксная форма:* $(a+b*c)*(c/d-e)$

2) Постфиксная форма: $abc*+cd/e-*$

Переменная	Значение
a	1
b	2
c	3
d	4
e	5

Стек вычисления:

		3			4	5		
	2	2	6	3	3	$\frac{3}{4}$	-4,25	
1	1	1	1	7	7	7	7	-29,75

3.2 Описание классов

3.2.1 Класс Stack

Объявление класса:

```
template <typename T> class Stack {
private:
    T* data;
    int maxSize;
    int top_;

    void resize(int step = 10);
public:
    Stack(int size = 10);
    Stack(const Stack<T>& stack);
    ~Stack();
    void pop();
    T top() const;
    void push(const T& element);
    bool isEmpty() const;
    bool isFull() const;
    const Stack<T>& operator=(const Stack<T>& stack);
    bool operator==(const Stack<T>& stack) const;
};
```

Поля:

data – память для представления стека.

top_ – индекс верхнего элемента стека.

maxSize – размер стека.

Конструкторы:

Stack(int size = 10)

Назначение: создание экземпляра класса.

Входные параметры: **size** – размер стека.

Выходные параметры: отсутствуют.

Stack(const Stack<T>& stack)

Назначение: создает копию существующего стека.

Входные параметры: **stack** – ссылка, адрес экземпляра класса, на основе которого будет создан другой.

Выходные параметры: отсутствуют.

Деструктор:

~Stack()

Назначение: очистка выделенной памяти.

Входные и выходные параметры отсутствуют.

Методы:

void resize(int step = 10)

Назначение: изменяет размер стека на заданный шаг.

Входные данные: **step** – шаг изменения размера.

Выходные данные: отсутствуют.

void pop()

Назначение: удаляет элемент с верхушки стека.

Входные данные: отсутствуют.

Выходные данные: отсутствуют.

T top() const

Назначение: возвращает элемент на верхушке стека.

Входные данные: отсутствуют.

Выходные данные: элемент, находящийся на вершине стека.

void push(const T& element)

Назначение: добавляет элемента на вершине стека.

Входные данные: **element** – добавляемый элемент.

Выходные данные: отсутствуют.

bool isEmpty(void) const

Назначение: проверка на пустоту стека.

Входные параметры: отсутствуют.

Выходные параметры: **true** – стек пуст, **false** в противном случае.

```
bool isFull (void) const
```

Назначение: проверка на полноту стека.

Входные параметры: отсутствуют.

Выходные параметры: **true** – стек заполнен, **false** в противном случае

```
const Stack<T>& operator=(const Stack<T>& stack)
```

Назначение: оператор присваивания.

Входные параметры: **stack** – стек, который мы присваиваем.

Выходные параметры: ссылка на экземпляр класса **Stack**.

3.2.2 Класс Parser

```
class Parser {
private:
    static bool isOperator(const string& token);
    static int getPrecedence(const string& op);
public:
    static bool isValidExpression(const vector<string>& infixExpression);
    static vector<string> splitExpression(const string& expression);
    static vector<string> infixToPostfix(
        const vector<string>& infixExpression);
    static map<string, double> getOperandValues(
        const vector<string>& tokens);
    static double evaluatePostfixExpression(const map<string,
        double>& operandValues,
        vector<string>& postfixExpression);
};
```

Методы:

```
static bool isOperator(const string& token)
```

Назначение: определяет, является ли заданный токен оператором.

Входные параметры: **token** – константная ссылка на строку, представляющую токен.

Выходные параметры: **true**, если **token** является оператором, в противном случае **false**.

```
static int getPrecedence(const string& op)
```

Назначение: возвращает приоритет оператора.

Входные параметры: **op** – константная ссылка на строку, представляющую оператор

Выходные параметры: **int** – приоритет оператора. Большее значение указывает на более высокий приоритет.

```
static bool isValidExpression(const vector<string>& infixExpression)
```

Назначение: проверяет, является ли данное инфиксное арифметическое выражение корректным.

Входные параметры: **infixExpression** – константная ссылка на вектор строк, представляющих инфиксное выражение.

Выходные параметры: **true**, если выражение корректно, иначе **false**.

```
static vector<string> splitExpression(const string& expression)
```

Назначение: разделяет арифметическое выражение на отдельные токены.

Входные параметры: **expression** – строка, представляющая арифметическое выражение.

Выходные параметры: **vector<string>** – вектор строк, содержащий токены арифметического выражения.

```
static vector<string> infixToPostfix(const vector<string>& infixExpression)
```

Назначение: преобразует инфиксное арифметическое выражение в постфиксное.

Входные параметры: **infixExpression** – константная ссылка на вектор строк, представляющих инфиксное выражение.

Выходные параметры: **vector<string>** – вектор строк, содержащий постфиксное представление арифметического выражения.

```
static map<string, double> getOperandValues(const vector<string>& tokens)
```

Назначение: получает значения операндов от пользователя.

Входные параметры: **tokens** – константная ссылка на вектор строк, представляющих токены арифметического выражения.

Выходные параметры: **map<string, double>** – карта, связывающая имена операндов с их значениями.

```
static double evaluatePostfixExpression(const map<string, double>& operandValues, vector<string>& postfixExpression)
```

Назначение: вычисляет значение постфиксного арифметического выражения.

Входные параметры: **operandValues** – константная ссылка на карту, связывающую имена операндов с их значениями, **postfixExpression** – вектор строк, содержащий постфиксное представление арифметического выражения.

Выходные параметры: **double** – результат вычисления постфиксного выражения.

Заключение

В результате проделанной работы были созданы два класса, **Stack** и **Parser**, предоставляющие возможность преобразования математических выражений из инфиксной записи в постфиксную. Также были получены практические навыки реализации этого алгоритма и работы с постфиксной формой.

Литература

1. Польская запись [https://ru.wikipedia.org/wiki/Польская_запись].
2. Польская нотация [<https://habr.com/ru/articles/596925>]

Приложения

Приложение А. Реализация класса Stack

```
template <typename T> Stack<T>::Stack(int size) {
    if (size <= 0) throw "maxSize must be bigger than 0";
    maxSize = size;
    top_ = -1;
    data = new T[maxSize];
}

template <typename T> Stack<T>::Stack(const Stack<T>& stack) {
    maxSize = stack.maxSize;
    top_ = stack.top_;
    data = new T[maxSize];
    for (int i = 0; i <= top_; i++) {
        data[i] = stack.data[i];
    }
}

template <typename T> Stack<T>::~~Stack() {
    if (data != NULL) {
        delete[] data;
        top_ = -1;
        maxSize = 0;
    }
}

template <typename T> void Stack<T>::resize(int step) {
    if (step <= 0) { throw "Step<0 or step=0"; }
    int newSize = maxSize + step;
    T* newData = new T[newSize];
    for (int i = 0; i < maxSize; ++i) {
        newData[i] = data[i];
    }
    delete[] data;
    data = newData;
    maxSize = newSize;
}

template <typename T> bool Stack<T>::isFull() const {
    return (top_ == maxSize - 1);
}

template <typename T> bool Stack<T>::isEmpty() const {
    return (top_ == -1);
}

template <typename T> T Stack<T>::top() const {
    if (isEmpty()) throw "Stack is empty";
    return data[top_];
}

template <typename T> void Stack<T>::push(const T& element) {
    if (isFull()) { resize(5); }
    data[++top_] = element;
}

template <typename T> void Stack<T>::pop() {
    if (isEmpty()) throw "Stack is empty!";
    top_--;
}
```



```

}

template <typename T> const Stack<T>& Stack<T>::operator=(const Stack<T>&
stack) {
    if (this != &stack) {
        delete[] data;
        maxSize = stack.maxSize;
        top_ = stack.top_;
        data = new T[maxSize];
        for (int i = 0; i <= top_; ++i) {
            data[i] = stack.data[i];
        }
    }
    return *this;
}

template <typename T> bool Stack<T>::operator==(const Stack<T>& stack) const
{
    if (top_ != stack.top_) {
        return false;
    }

    for (int i = 0; i < top_; ++i) {
        if (data[i] != stack.data[i]) {
            return false;
        }
    }

    return true;
}

```

Приложение Б. Реализация класса Parser

```

vector<string> Parser::splitExpression(const string& expression) {
    vector<string> result;
    string currentToken;

    for (char ch : expression) {
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '(' ||
ch == ')') {
            if (!currentToken.empty()) {
                result.push_back(currentToken);
                currentToken.clear();
            }
            result.push_back(string(1, ch));
        }
        else if (isspace(ch)) {
            continue;
        }
        else {
            currentToken += ch;
        }
    }
    if (!currentToken.empty()) {
        result.push_back(currentToken);
    }
    return result;
}

int Parser::getPrecedence(const string& op) {
    if (op == "+" || op == "-") {
        return 1;
    }
}

```

```

        else if (op == "*" || op == "/") {
            return 2;
        }
        return 0;
    }
    bool Parser::isOperator(const string& token) {
        return (token == "+" || token == "-" || token == "*" || token == "/");
    }

    bool Parser::isValidExpression(const vector<string>& infixExpression) {
        if (isOperator(infixExpression[0])) return 0;
        else if (isOperator(infixExpression[infixExpression.size() - 1])) return
0;
        else{
            for (int i = 0; i < infixExpression.size() - 1; i++) {
                if(isOperator(infixExpression[i])                                &&
isOperator(infixExpression[i+1])) return 0;
                if(infixExpression[i] == "(" && isOperator(infixExpression[i +
1])) return 0;
            }
        }
        return 1;
    }

    map<string, double> Parser::getOperandValues(const vector<string>& tokens) {
        map<string, double> operandValues;

        for (const auto& token : tokens) {
            if (token != "+" && token != "-" && token != "*" && token != "/" &&
token != "(" && token != ")") {
                if (operandValues.find(token) == operandValues.end()) {
                    double value;
                    if (token.find_first_not_of("0123456789.") == string::npos) {
                        operandValues[token] = stod(token);
                        continue;
                    }
                    cout << "Enter operand value " << token << ": ";
                    cin >> value;
                    operandValues[token] = value;
                }
            }
        }
        return operandValues;
    }

    vector<string> Parser::infixToPostfix(const vector<string>& infixExpression)
{
        Stack<string> operands;
        Stack<string> operators;

        for (const string& token : infixExpression) {
            if (isalnum(token[0])) {
                operands.push(token);
            }
            else if (isOperator(token)) {
                while (!operators.isEmpty() && getPrecedence(operators.top()) >=
getPrecedence(token)) {
                    operands.push(operators.top());
                    operators.pop();
                }
                operators.push(token);
            }
        }
    }

```

```

        else if (token == "(") {
            operators.push(token);
        }
        else if (token == ")") {
            while (!operators.isEmpty() && operators.top() != "(") {
                operands.push(operators.top());
                operators.pop();
            }
            operators.pop();
        }
    }

    while (!operators.isEmpty()) {
        operands.push(operators.top());
        operators.pop();
    }

    vector<string> postfix;

    while (!operands.isEmpty()) {
        postfix.push_back(operands.top());
        operands.pop();
    }
    reverse(postfix.begin(), postfix.end());
    return postfix;
}

double Parser::evaluatePostfixExpression(const map<string, double>&
operandValues, vector<string>& postfixExpression) {
    Stack<double> resultStack;

    for (int i = 0; i < postfixExpression.size(); i++) {
        const string token = postfixExpression[i];

        if (!(isOperator(token))) {
            if (operandValues.find(token) != operandValues.end()) {
                resultStack.push(operandValues.at(token));
            }
            else {
                throw invalid_argument("Error: operands value not exist.\n");
                return 0.0;
            }
        }
        else {
            double operand2 = resultStack.top();
            resultStack.pop();

            double operand1 = resultStack.top();
            resultStack.pop();

            if (token == "+") {
                resultStack.push(operand1 + operand2);
            }
            else if (token == "-") {
                resultStack.push(operand1 - operand2);
            }
            else if (token == "*") {
                resultStack.push(operand1 * operand2);
            }
            else if (token == "/") {
                if (operand2 == 0) {
                    throw "ERROR: divizion by zero.\n";
                    return 0.0;
                }
            }
        }
    }
}

```

```

        }
        resultStack.push(operand1 / operand2);
    }
}
}
return resultStack.top();
}

```

Приложение В. Реализация класса sample_parser

```

int main() {
    string inputExpression;
    cout << "Enter expression: ";
    getline(cin, inputExpression);

    vector<string> tokens = Parser::splitExpression(inputExpression);

    if (!(Parser::isValidExpression(tokens))) {
        cerr << "ERROR: Invalid Expression!";
        return 1;
    }

    vector<string> res = Parser::infixToPostfix(tokens);
    for (int i = 0; i < res.size(); i++) cout << res[i] << ", ";
    cout << endl;

    map<string, double> opVal = Parser::getOperandValues(tokens);
    try{
        cout << Parser::evaluatePostfixExpression(opVal, res);
    }
    catch (const char* error){
        cout << error;
    }
    return 0;
}

```