Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное   
учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«Вычисление арифметических выражений»**

**Выполнил**:

студент группы 382003-1

Лапин Дмитрий

**Проверил**:

ассистент каф. МОСТ,

Арисова А.Н.

Нижний Новгород

2021

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

# Постановка задачи

Требуется написать программу на языке C++, которая вычисляет реализует калькулятор арифметических выражений, которые передает туда пользователь. Программа переводит выражение в обратную польскую запись и считает результат выражение через нее. Допустимые операции: +, -, /, \* в выражении, и унарный минус. Допускается использование скобок. Предполагается, что вычисления, преобразования будут производиться с использованием структуры данных Стек.

Стек – структура данных, представляющий из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Говорят, что стек устроен по принципу FILO(first in - last out, «первым пришел – последним вышел»).

# Метод решения

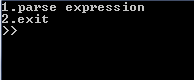
1. Проверка корректности выражения

* корректность ввода
* пропущены ли операнды или знаки операций
* недопустимые символы

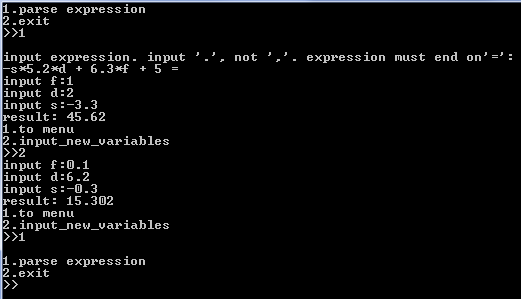
1. Разбиение исходного арифметического выражения на лексемы (т.е. выделить операнды, операции, переменные). В процессе выражение проверяется на корректность.
2. Перевод выражения в постфиксную (польскую) запись
3. Вычисление выражения по постфиксной записи

# Руководство пользователя

При запуске программы пользователю предлагается меню, состоящее из 2 пунктов.



Затем, пользователю нужно ввести выражение. Переменные в выражении состоят только из латинских символов, запятая соответствует точке, выражение обязательно должно заканчиваться на ‘=’. Далее программа предоставит ему поля для ввода переменных, которые пользователь ввел в выражении. Если выражение введено некорректно, то пользователь увидит исключение, которое выбросит проверяющая функция. Далее пользователь может посчитать выражение повторно, но с другими значениями переменных или выйти в меню.



# Описание программной реализации

Класс Parser отвечает за парсинг строки, ее трансформацию в обратную польскую запись и вычисление значения числа. Функция ParseToPolish() отвечает за переработку введенной пользователем строчки в стэк – результат алгоритма обратной польской записи.

Функция inputVariables() отвечает за ввод пользователем переменных, которые есть в выражении.

Функция CountPolish() отвечает за подсчет значения выражения, на основе стэка, полученного из ParseToPolish().

Функция ExamInCorrect проверяет число на корректность ввода. В случае некорректного ввода функция выбрасывает исключение.

Так же в классе есть переменная логического типа IsContainVariables, которая отвечает за то, есть ли в выражении пользовательские переменные или нет.

# Приложение

void ParseToPolish() {

expr.erase(remove\_if(expr.begin(), expr.end(), isspace), expr.end());

ExamOnInCorrect();

ExamOnContainVariables();

std::string x; std::string t;

TStack <std::string> OperationStack;

char ch = '+';

int pos = 0;

do {

ch = expr[pos];

x = std::string(1, ch);

if (('A' <= ch && ch <= 'Z') || ('a' <= ch && ch <= 'z')) {

std::string t = "";

do {

t += ch;

pos += 1;

ch = expr[pos];

} while (IsSymbol(ch));

PolishStack.push(t);

}

else if (('0' <= ch && ch <= '9') || ch == '.') {

std::string t = "";

int count = 0;

do {

t += ch;

pos += 1;

ch = expr[pos];

if (ch == '.') count += 1;

} while (('0' <= ch && ch <= '9') || ch == '.');

if (count > 1) { std::cout << "incorrect\_input\_of\_float\_numbers"; throw std::invalid\_argument("incorrect\_input\_of\_float\_numbers"); }

PolishStack.push(t);

}

else if (x == "(") {

OperationStack.push(x); pos += 1;

}

else if (x == ")") {

pos += 1;

while (1) {

std::string t = OperationStack.top();

OperationStack.pop();

if (t == "(") break;

PolishStack.push(t);

}

}

else if ((pos == 0 && ch == '-') || (ch == '-' && IsOperation(expr[pos - 1])) || (ch == '-' && expr[pos - 1] == '(')) {

pos += 1;

ch = '@';

x = "@";

while (!OperationStack.empty()) {

std::string t = OperationStack.top();

if (GetOperationPrt(x) < GetOperationPrt(t)) {

OperationStack.pop();

PolishStack.push(t);

}

else {

break;

}

}

OperationStack.push(x);

}

else if (IsOperation(ch)) {

pos += 1;

while (!OperationStack.empty()) {

std::string t = OperationStack.top();

if (GetOperationPrt(x) <= GetOperationPrt(t)) {

OperationStack.pop();

PolishStack.push(t);

}

else {

break;

}

}

OperationStack.push(x);

}

else { throw std::invalid\_argument("user\_input\_unknown\_symbol"); }

} while (x != "=");

}

void inputVariables() {

TStack <std::string> temp\_stack(PolishStack);

while (!temp\_stack.empty()) {

std::string lem = temp\_stack.top();

if (IsSymbol(lem[0])) {

std::cout << "input " << lem << ":";

std::cin >> lem;

for (int i = 0; i < lem.size(); i++) {

if (!(IsAlpha(lem[i]) || lem[i] == '.' || lem[i] == '-')) {

std::cout << "incorrect\_input\_of\_variables";

throw std::invalid\_argument("incorrect\_input\_of\_variables");

}

}

}

resultParse.push(lem);

temp\_stack.pop();

}

}

float CountPolish() {

TStack<std::string> count;

while (!resultParse.empty()) {

std::string temp = resultParse.top();

resultParse.pop();

if (('0' <= temp[0] && temp[0] <= '9') || (temp[0] == '-' && temp.size() > 1))

{

count.push(temp);

}

else if (temp[0] == '@') {

std::string first = count.top();

count.pop();

std::string sh = std::to\_string(-(std::stof(first)));

count.push(sh);

}

else if (IsOperation(temp[0])) {

std::string second = count.top();

count.pop();

std::string first = count.top();

count.pop();

std::string sh = std::to\_string(doOperation(std::stof(first), std::stof(second), temp[0]));

count.push(sh);

}

}

return std::stof(count.top());

}

void ExamOnInCorrect() {

int count = 0;

for (int i = 0; i < expr.size(); i++) {

if (expr[i] == '(') count++;

if (expr[i] == ')') count--;

}

if (count != 0)

throw std::invalid\_argument("num\_of\_closing\_and\_opening\_brackets\_not\_equal");

for (int i = 0; i < expr.size(); i++)

{

if (i == 0) {

if (IsOperation(expr[i]) && expr[i] != '=' && expr[i] != '-') {

std::cout << "binar\_operation\_cannot\_be\_first\_symbol\_of\_equation";

throw std::invalid\_argument("binar\_operation\_cannot\_be\_first\_symbol\_of\_equation");

}

else if (expr[i] == ')') { std::cout << "closed\_bracket\_cannot\_start\_the\_expression "; throw std::invalid\_argument("close\_bracket\_cannot\_start\_the\_expression "); }

else if (expr[i] == '.') { std::cout << "dot\_cannot\_start\_the\_expression"; throw std::invalid\_argument("dot\_cannot\_start\_the\_expression "); }

else if (expr[i] == '(' && IsOperation(expr[i + 1]) && expr[i + 1] != '-') {

std::cout << "binar\_operator\_after\_opening\_bracket";

throw std::invalid\_argument("binar\_operator\_after\_opening\_bracket");

}

}

else {

if (IsOperation(expr[i - 1]) && IsOperation(expr[i]) && expr[i] != '-') {

std::cout << "two binar operations cannot be one after another";

throw std::invalid\_argument("two binar operations cannot be one after another");

}

else if (expr[i] == ')' && IsOperation(expr[i - 1])) {

std::cout << "operator\_before\_closing\_bracket";

throw std::invalid\_argument("operator\_before\_closing\_bracket");

}

else if (expr[i] == ')' && (IsAlpha(expr[i + 1]) || IsSymbol(expr[i + 1]))) {

std::cout << "symbol\_or\_alpha\_after\_closing\_bracket";

throw std::invalid\_argument("symbol\_or\_alpha\_after\_closing\_bracket");

}

else if (expr[i] == '(' && IsOperation(expr[i + 1]) && expr[i + 1] != '-') {

std::cout << "binar\_operator\_after\_opening\_bracket";

throw std::invalid\_argument("binar\_operator\_after\_opening\_bracket");

}

else if (expr[i] == '(' && (IsAlpha(expr[i - 1]) || IsSymbol(expr[i - 1]))) {

std::cout << "symbol\_or\_alpha\_before\_opening\_bracket";

throw std::invalid\_argument("symbol\_or\_alpha\_before\_opening\_bracket");

}

else if (expr[i] != ')' && expr[i] != '(' && !IsSymbol(expr[i]) && !IsAlpha(expr[i]) && !IsOperation(expr[i]) && expr[i] != '.') {

std::cout << "unknown symbol";

throw std::invalid\_argument("unknown\_symbol");

}

else if (expr[i] != ')' && expr[i] != '(' && !IsSymbol(expr[i]) && !IsAlpha(expr[i]) && !IsOperation(expr[i]) && expr[i] != '.') {

std::cout << "unknown symbol";

throw std::invalid\_argument("unknown\_symbol");

}

else if ((expr[i] == '.' && !IsAlpha(expr[i - 1]) && !IsAlpha(expr[i + 1])) || (expr[i] == '.' && IsAlpha(expr[i - 1]) && !IsAlpha(expr[i + 1])) || (expr[i] == '.' && !IsAlpha(expr[i - 1]) && IsAlpha(expr[i + 1]))) { std::cout << "dot\_without\_numbers"; throw std::invalid\_argument("dot\_without\_numbers"); }

}

}

if (expr[expr.size() - 1] != '=') {

std::cout << "equal\_not\_last\_symbol\_of\_equation";

throw std::invalid\_argument("equal\_not\_last\_symbol\_of\_equation");

}

}