Федеральное государственное автономное образовательное учреждение

высшего образования

«Нижегородский государственный университет

им. Н.И. Лобачевского»

Факультет прикладной математики и информатики

Отчёт по лабораторной работе

Вычисление арифметических выражений

Выполнил:

студент ф-та ПМИ гр. 381703-1

Боганов С. В.

Проверил:

Волокитин В. Д.

Нижний Новгород

2018 г.

**Содержание**

[Введение](#Введение) 3

[Постановка задачи](#Постановка_задачи) 4

[Руководство пользователя](#Руководство_пользователя) 5

[Руководство программиста](#Руководство_программиста) 6

[Описание структур данных](#Описание_структур_данных) 6

[Описание алгоритмов](#Описание_алгоритмов) 8

[Описание структуры программы](#Описание_структуры_программы) 10

[Заключение](#Заключение) 12

[Литература](#Литература) 13

[Приложения](#Приложения) 14

**Введение**

Арифметические выражения кажутся простыми лишь на первый взгляд. Проблемы начинаются тогда, когда мы ставим перед собой задачу “научить” компьютер их вычислять. Для этой задачи существует прекрасный механизм, известный, как обратная польская запись, названный в честь польского логика Я. Лукасевича, изучавшего его свойства. Именно этот механизм реализован в моей лабораторной работе.

**Постановка задачи**

Необходимо разработать и реализовать программу, выполняющую вычисление арифметического выражения с вещественными числами.

**Требования к программе:**

* Программа должна быть написана на языке C++.
* Выражение в качестве операндов может содержать переменные и вещественные числа.
* Допустимые операции: +, -, /, \*.
* Допускается наличие знака "-" в начале выражения или после открывающей скобки.
* Все функции должны тестироваться с помощью Google tests.
* Разработка шаблонного класса TStack.
* Разбиение исходного арифметического выражения на лексемы (т.е. выделение операндов, операций и переменных)
* Проверка корректности выражения:
  + Правильность расстановки скобок (с использованием стека (TStack) для хранения индексов скобок).
  + Проверка на корректность расположения операндов и операций относительно друг друга.
  + Проверка на недопустимые символы.
* Программа должна сообщать пользователю вид ошибки и номера символов строки, в которых были найдены ошибки.
* Перевод выражения в постфиксную (польскую) запись.
* Вычисление выражения по постфиксной записи.
* Создание консольного интерфейса пользователя.

**Руководство пользователя**

После запуска программы, пользователь должен ввести арифметическое выражение, согласно стандартным правилам ввода. Вещественные числа следует разделять точкой. Кроме этого, пользователь может использовать любой из 26 параметров: a, …, z, с их последующей инициализацией. Все пробелы игнорируются.

На выходе пользователь получает вычисленное значение выражения или сообщение о виде ошибки и номер символа строки, в котором она была обнаружена.

**Руководство программиста**

**Описание структур данных**

**main\_arithmetic.cpp**

**bool** key1 - переменная для обработки повторного ввода выражения.

**string** str - строка с математическим выражением.

**double** ans - переменная с вычисленным значением выражения.

**arithmetic.cpp**

**int** i. k - переменные для управления циклом.

**double**

t, t1, t2 - переменные, предназначенные для получения операндов из стека.

ans - переменная с вычисленным значением выражения.

**string**

temp - временная переменная.

s - строка, полученная из исходной после удаления пробелов.

**TLexeme**

p, p0, temp - временные переменные.

**TStack**<**double**> s - стек, необходимый для вычисления выражения в обратной польской записи.

**pair**<**int**, **int**> err - пара чисел, содержащая ошибку и номер символа строки, в котором она найдена.

**map**<**char**, **TStack**<**int**>> parameters - map, содержащий переменные и номера символов строки, в которых они найдены.

**vector**<**pair**<**TLexeme**, **int**>>

v - вектор, содержащий лексему и номер символа строки, в котором она начинается.

ans - вектор, полученный после перевода вектора лексем в вектор лексем, расположенных соответственно обратной польской записи.

**pair**<**bool**, **int**> temp - пара чисел, содержащая информацию о том, обнаружена ли ошибка в использовании унарного минуса и тип такой ошибки.

**map**<**pair**<**char**, **int**>, **int**> m - map, содержащий операцию, её тип (бинарная или унарная) и приоритет.

**TStack**<**pair**<**TLexeme**, **int**>> s - стек, необходимый для перевода вектора лексем в обратную польскую запись.

**map**<**int**, **vector**<**int**>> m - map, содержащий информацию о том, какие лексемы могут идти после заданной лексемы.

**Описание алгоритмов**

Код функций см. в приложении.

**arithmetic.cpp**

Конвертация строки в вектор лексем

1. Удаляем пробелы из строки.
2. Проверяем поочередно каждый символ строки:
   1. Проверяем, является ли символ унарной операцией.
   2. Проверяем, является ли символ бинарной операцией.
   3. Проверяем, является ли последовательность символов числом.
   4. Проверяем, является ли символ параметром.
3. Если символ (или последовательность символов) соответствует одному из вышеперечисленных типов, добавляем его (её) в вектор лексем.
4. Если символ (или последовательность символов) не соответствует ни одному из вышеперечисленных типов, выбрасываем исключение.
5. После поочередной проверки каждого символа лексем просим пользователя задать параметры, после чего вместо самих параметров в вектор лексем записываем числа, введённые пользователем.

Проверка на корректность расположения операндов и операций

1. Проверяем первую лексему. Если она является закрывающейся скобкой или бинарной операцией, выбрасываем исключение.
2. Проверяем последнюю лексему. Если она является унарной / бинарной операцией, открывающейся скобкой, выбрасываем исключение.
3. Создаём стек для проверки правильности расстановки скобок.
4. Рассматриваем поочередно каждую лексему:
   1. Если лексема является отрывающейся скобкой, кладем её в стек.
   2. Если лексема является закрывающейся скобкой, то:
      * 1. Если стек не пустой и верхний элемент является открывающейся скобкой, то удаляем её.
        2. Иначе выбрасываем исключение.
   3. Проверяем, может ли следующая лексема стоять после исходной. Если может, переходим к рассмотрению следующей лексемы, если же нет - выбрасываем исключение.
5. Если стек, в который мы складывали скобки не пустой, то выбрасываем исключение.

Перевод в обратную польскую запись

Рассматриваем поочередно каждую лексему:

1. Если лексема является числом, помещаем её в выходную строку.
2. Если лексема является операцией, то проверяем её приоритет. Унарный минус имеет приоритет 3, умножение и деление - 2, сложение и вычитание - 1, а открывающаяся скобка - 0. Получив одну из этих операций, необходимо проверить стек:
   1. Если стек все еще пуст, или находящиеся в нем лексемы имеют меньший приоритет, чем приоритет текущей лексемы, то помещаем текущую лексему в стек.
   2. Если лексема, находящаяся на вершине стека имеет приоритет, больший или равный приоритету текущей лексемы, то извлекаем лексемы из стека в выходную строку до тех пор, пока выполняется это условие. Затем переходим к предыдущему пункту (а).
3. Если лексема является открывающейся скобкой, то помещаем её в стек.
4. Если лексема является закрывающейся скобкой, то извлекаем лексемы из стека в выходную строку до тех пор, пока не встретим в стеку открывающуюся скобку, которую следует удалить из стека. Закрывающуюся скобку тоже следует удалить.

Если вся входная строка разобрана, а в стеке еще остаются знаки операций, извлекаем их из стека в выходную строку.

Вычисление выражения по обратной польской записи

Рассматриваем поочередно каждую лексему:

1. Если лексема является числом, то кладем её в стек.
2. Если лексема является унарной (бинарной) операцией, то извлекаем из стека верхнее число (два верхних числа), используем его (их) в качестве операнда (операндов) для этой операции, после чего кладем результат обратно в стек.

Когда весь входной вектор лексем будет разобран, в стеке останется одно число, которое будет результатом данного выражения.

**Описание структуры программы**

Программа состоит из 4-х проектов: **arithmetic**, **gtest**, **sample** и **tests**.

Проект **gtest** включает в себя 2 файла gtest.h и gtest-all.cc, которые содержат реализацию функционала google tests.

Проект **tests** включает в себя 3 файла: test\_main.cpp, test\_stack.cpp, test\_arithmetic.cpp.

* Файл test\_main.cpp запускает все google tests.
* Файл test\_stack.cpp содержит тесты для методов класса TStack.
* Файл test\_arithmetic.cpp содержит тесты для методов класса TLexeme и функций, предназначенных для вычисления значения выражения.

Проект **sample** включает в себя файл main\_arithmetic.cpp, который содержит реализацию пользовательского приложения, предназначенного для вычисления значения арифметических выражений.

Проект **arithmetic** включает в себя 3 файла: arithmetic.h, stack.h и arithmetic.cpp.

* Файл stack.h содержит шаблонный класс TStack.

Класс TStack содержит:

* + 3 поля: size (размер стека), top (вершина стека), вектор v (содержит элементы стека).
  + Прототипы и реализацию 10 методов: конструктора по умолчанию и конструктора инициализации, деструктора, проверки на полноту и пустоту (проверка на полноту имеет спецификатор доступа private), получения текущего количества элементов в стеке, вставки и извлечения элемента, просмотра верхнего элемента и очистки стека.
* Файл arithmetic.h содержит 3 перечисления (type, sigh и errors), структуру elem\_or\_oper, класс TLexeme, а также прототипы следующих функций: Calculate, Solver, Converting\_number, Check\_number, Create\_lexeme\_array, Create\_RPN\_array, New\_line\_without\_spaces, Error\_checking, Type\_checking, Error\_output, Set\_parameters и Check\_unary\_minus.

Перечисление type содержит тип лексемы: число, бинарная операция, унарная операция, открывающаяся скобка, закрывающаяся скобка и признак того, что лексема не является ни одним из вышеперечисленных типов.

Перечисление sign содержит признак того, положительное число или отрицательное.

Перечисление errors содержит виды ошибок: деление на ноль, некорректная точка в числе, неизвестный символ, некорректный первый символ, некорректный последний символ, неправильная скобочная последовательность и пропуск операции или операнда.

Структура elem\_or\_oper содержит 2 поля: элемент и операция.

Класс TLexeme содержит:

* + 2 поля: value (содержит операцию или число), type\_elem (содержит тип лексемы).
  + Прототипы 8 методов: конструктора по умолчанию, 2-х конструкторов инициализации (для числа и операции), конструктора копирования, оператора присваивания, деструктора, возвращения значения лексемы и возвращения типа лексемы.
* Файл arithmetic.cpp содержит реализацию методов класса TLexeme и функций, объявленных в файле arithmetic.h.

**Заключение**

Достигнутые результаты:

* Разработана и реализована программа, выполняющая вычисление арифметического выражения с вещественными числами.
* Программа написана на языке C++.
* Выражение в качестве операндов может содержать переменные и вещественные числа, а в качестве операций +, -, /, \*.
* Реализован унарный минус.
* Программа выполняет предварительную проверку корректности выражения и сообщает пользователю вид ошибки и номера символов строки, в которых были найдены ошибки.
* Все функции тестируются с помощью Google tests.
* Разработан шаблонный класс TStack.
* Исходное арифметическое выражение разбивается на лексемы.
* Перевод выражения в постфиксную (польскую) запись.
* Вычисление выражения по постфиксной (польской) записи.
* Создан консольный интерфейс пользователя.

**Литература**

1. Алгоритм преобразования выражения в ОПЗ и его последующего вычисления.

[http://trubetskoy1.narod.ru/ppn.html]

**Приложения**

**Проект arithmetic**

**Файл stack.h**

Класс TStack

const int size0 = 50;

template <class ValType>

class TStack

{

private:

int top;

int size;

std::vector<ValType> v;

bool isFull() const;

public:

TStack(int n = size0);

virtual ~TStack() {};

bool isEmpty() const;

void push(ValType elem);

int CurrentSize() const;

ValType front() const;

ValType pop();

void AllClear();

};

Конструктор инициализации / конструктор по умолчанию

template <class ValType>

TStack<ValType>::TStack(int n)

{

if (n <= 0)

{

throw "Incorrect length";

}

size = n;

v.resize(size);

top = -1;

}

Получение текущего количества элементов в стеке

template <class ValType>

int TStack<ValType>::CurrentSize() const

{

return top + 1;

}

Просмотр верхнего элемента стека

template <class ValType>

ValType TStack<ValType>::front() const

{

if (isEmpty())

{

throw "Stack is empty";

}

return v[top];

}

Проверка стека на пустоту

template <class ValType>

bool TStack<ValType>::isEmpty() const

{

return top == -1;

}

Проверка стека на полноту (private)

template <class ValType>

bool TStack<ValType>::isFull() const

{

return top == size - 1;

}

Вставка элемента

template <class ValType>

void TStack<ValType>::push(ValType elem)

{

if (isFull())

{

v.resize(size \* 2);

size = size \* 2;

}

v[++top] = elem;

}

Удаление верхнего элемента стека

template <class ValType>

ValType TStack<ValType>::pop()

{

if (isEmpty())

{

throw "Stack is empty";

}

return v[top--];

}

Удаление всех элементов стека

template <class ValType>

void TStack<ValType>::AllClear()

{

size = size0;

top = -1;

v.clear();

v.resize(size);

}

**Файл arithmetic.h**

Перечисление sign

enum sign

{

positive,

negative

};

Перечисление type

enum type

{

number,

binary\_operation,

unary\_operation,

op\_bracket,

cl\_bracket,

no\_type

};

Перечисление errors

enum errors

{

division\_by\_zero,

incorrect\_point,

unknown\_symbol,

incorrect\_first\_symbol,

incorrect\_last\_symbol,

wrong\_bracket\_sequence,

missed\_operation\_or\_operand

};

Структура, содержащая элемент и операцию

struct elem\_or\_oper

{

double elem;

char oper;

};

Класс TLexeme

class TLexeme

{

private:

elem\_or\_oper value;

int type\_elem;

public:

TLexeme();

TLexeme(char, int t = binary\_operation);

TLexeme(double);

TLexeme(const TLexeme&);

TLexeme& operator=(const TLexeme&);

~TLexeme() {};

elem\_or\_oper GetValue() const;

int GetType() const;

};

**Файл arithmetic.cpp**

Конструктор по умолчанию

TLexeme::TLexeme()

{

type\_elem = no\_type;

}

Конструктор инициализации для операции

TLexeme::TLexeme(char ch, int p)

{

if ((p != binary\_operation && p != unary\_operation) || (ch != '\*' && ch != '/' && ch != '+' && ch != '-' && ch != ')' && ch != '('))

{

throw "Incorrect initialization";

}

else if (p == binary\_operation && (ch == '\*' || ch == '/' || ch == '+' || ch == '-'))

{

type\_elem = binary\_operation;

value.oper = ch;

}

else if (p == unary\_operation && ch == '-')

{

type\_elem = unary\_operation;

value.oper = ch;

}

else if (ch == '(')

{

type\_elem = op\_bracket;

value.oper = ch;

}

else if (ch == ')')

{

type\_elem = cl\_bracket;

value.oper = ch;

}

}

Конструктор инициализации для числа

TLexeme::TLexeme(double a)

{

value.elem = a;

type\_elem = number;

}

Конструктор копирования

TLexeme::TLexeme(const TLexeme& a)

{

value.elem = a.value.elem;

value.oper = a.value.oper;

type\_elem = a.type\_elem;

}

Оператор присваивания

TLexeme& TLexeme::operator=(const TLexeme& a)

{

value.elem = a.value.elem;

value.oper = a.value.oper;

type\_elem = a.type\_elem;

return \*this;

}

Функция, возвращающая значение лексемы

elem\_or\_oper TLexeme::GetValue() const

{

return value;

}

Функция, возвращающая тип лексемы

int TLexeme::GetType() const

{

return type\_elem;

}

Вычисление выражения по обратной польской записи

double Solver(const vector<pair<TLexeme, int>> &v)

{

TStack<double> s;

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

{

TLexeme temp = v[i].first;

if (temp.GetType() == number)

{

s.push(temp.GetValue().elem);

}

else if (temp.GetType() == unary\_operation)

{

double t = s.pop();

double ans;

if (temp.GetValue().oper == '-')

{

ans = -t;

}

s.push(ans);

}

else if (temp.GetType() == binary\_operation)

{

double t2 = s.pop(), t1 = s.pop();

double ans;

switch (temp.GetValue().oper)

{

case '+':

ans = t1 + t2;

break;

case '-':

ans = t1 - t2;

break;

case '\*':

ans = t1 \* t2;

break;

case '/':

if (abs(t2) < 1e-7)

{

pair<int, int> err(division\_by\_zero, v[i].second);

throw err;

}

ans = t1 / t2;

break;

}

s.push(ans);

}

}

return s.pop();

}

Конвертация строки в число

double Сonverting\_number(const string& s, int index, int sign)

{

int k = 0;

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

{

if (s[i] == '.')

{

k++;

}

if (k > 1)

{

pair<int, int> err(incorrect\_point, index + i);

throw err;

}

}

double ans;

if (s[0] == '.')

{

pair<int, int> err(incorrect\_point, index);

throw err;

}

else if (s[s.size() - 1] == '.')

{

pair<int, int> err(incorrect\_point, index + s.size() - 1);

throw err;

}

else

{

ans = atof(s.c\_str());

if (sign == negative)

{

ans = -ans;

}

}

return ans;

}

Проверка числа на корректность

TLexeme Check\_number(const string &s, int &i, int sign)

{

string temp = "";

int pos = i, k = i;

while ((s[k] >= '0' && s[k] <= '9') || s[k] == '.')

{

temp += s[k];

k++;

}

i = k - 1;

TLexeme p(Сonverting\_number(temp, pos, sign));

return p;

}

Удаление пробелов из строки

string New\_line\_without\_spaces(const string &s)

{

string temp = "";

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

{

if (s[i] != ' ')

{

temp += s[i];

}

}

return temp;

}

Проверка унарного минуса на корректность

pair<bool, int> Check\_unary\_minus(const string& s, int &i)

{

if (i == 0)

{

if (s[0] == '-' && s.size() > 1)

{

if (s[1] >= '0' && s[1] <= '9')

{

i++;

return make\_pair(true, 1);

}

else if (s[1] == '(')

{

return make\_pair(true, 2);

}

else if (s[1] >= 'a' && s[1] <= 'z')

{

return make\_pair(true, 5);

}

else if (s[1] == '-')

{

int k = 0;

while (s[i] == '-')

{

k++;

i++;

}

if (s[i] >= '0' && s[i] <= '9')

{

if (k & 1)

{

return make\_pair(true, 1);

}

else

{

return make\_pair(true, 0);

}

}

else if (s[i] == '(')

{

if (k & 1)

{

return make\_pair(true, 3);

}

else

{

return make\_pair(true, 4);

}

}

else if (s[i] >= 'a' && s[i] <= 'z')

{

if (k & 1)

{

return make\_pair(true, 6);

}

else

{

return make\_pair(true, 7);

}

}

}

}

return make\_pair(false, -1);

}

else if (i + 1 < s.size())

{

if ((s[i - 1] == '(' || s[i - 1] == '+' || s[i - 1] == '-' || s[i - 1] == '\*' || s[i - 1] == '/') && s[i] == '-' && (s[i + 1] >= '0' && s[i + 1] <= '9'))

{

i++;

return make\_pair(true, 1);

}

else if ((s[i - 1] == '(' || s[i - 1] == '+' || s[i - 1] == '-' || s[i - 1] == '\*' || s[i - 1] == '/') && (s[i] == '-' && s[i + 1] == '('))

{

return make\_pair(true, 2);

}

else if (s[i] == '-' && s[i + 1] == '(')

{

return make\_pair(true, 11);

}

else if ((s[i - 1] == '(' || s[i - 1] == '+' || s[i - 1] == '-' || s[i - 1] == '\*' || s[i - 1] == '/') && s[i] == '-' && (s[i + 1] >= 'a' && s[i + 1] <= 'z'))

{

return make\_pair(true, 5);

}

else if ((s[i - 1] == '(' || s[i - 1] == '+' || s[i - 1] == '-' || s[i - 1] == '\*' || s[i - 1] == '/') && (s[i] == '-' && s[i + 1] == '-'))

{

int k = 0;

while (s[i] == '-')

{

k++;

i++;

}

if (s[i] >= '0' && s[i] <= '9')

{

if (k & 1)

{

return make\_pair(true, 1);

}

else

{

return make\_pair(true, 0);

}

}

else if (s[i] == '(')

{

if (k & 1)

{

return make\_pair(true, 3);

}

else

{

return make\_pair(true, 4);

}

}

else if (s[i] >= 'a' && s[i] <= 'z')

{

if (k & 1)

{

return make\_pair(true, 6);

}

else

{

return make\_pair(true, 7);

}

}

}

else if (s[i] == '-' && s[i + 1] == '-')

{

int k = 0;

while (s[i] == '-')

{

k++;

i++;

}

if (s[i] >= '0' && s[i] <= '9')

{

if (k & 1)

{

return make\_pair(true, 8);

}

else

{

return make\_pair(true, 0);

}

}

else if (s[i] == '(')

{

if (k & 1)

{

return make\_pair(true, 9);

}

else

{

return make\_pair(true, 4);

}

}

else if (s[i] >= 'a' && s[i] <= 'z')

{

if (k & 1)

{

return make\_pair(true, 10);

}

else

{

return make\_pair(true, 7);

}

}

}

else

{

return make\_pair(false, -1);

}

}

else

{

return make\_pair(false, -1);

}

return make\_pair(false, -1);

}

Инициализация параметров

void Set\_parameters(const map<char, TStack<int>> &m, vector<pair<TLexeme, int>>& v, const vector<double>& input)

{

if (!input.empty())

{

if (input.size() != m.size())

{

throw "Incorrect initialization with parameters.";

}

else

{

int i = 0;

for (auto it : m)

{

TLexeme p(input[i++]);

if (it.second.CurrentSize() == 1)

{

v[it.second.front()] = make\_pair(p, it.second.front());

it.second.pop();

}

else

{

while (!it.second.isEmpty())

{

v[it.second.front()] = make\_pair(p, it.second.front());

it.second.pop();

}

}

}

}

}

else

{

for (auto it : m)

{

cout << it.first << " = ";

string temp;

cin >> temp;

int i = 0;

if (temp[i] == '-')

{

i++;

TLexeme p = Check\_number(temp, i, negative);

if (it.second.CurrentSize() == 1)

{

v[it.second.front()] = make\_pair(p, it.second.front());

it.second.pop();

}

else

{

while (!it.second.isEmpty())

{

v[it.second.front()] = make\_pair(p, it.second.front());

it.second.pop();

}

}

}

else

{

TLexeme p = Check\_number(temp, i, positive);

if (it.second.CurrentSize() == 1)

{

v[it.second.front()] = make\_pair(p, it.second.front());

it.second.pop();

}

else

{

while (!it.second.isEmpty())

{

v[it.second.front()] = make\_pair(p, it.second.front());

it.second.pop();

}

}

}

}

}

}

Конвертация строки в вектор лексем

vector<pair<TLexeme, int>> Create\_lexeme\_array(const string& str, const vector<double>& input)

{

map<char, TStack<int>> parameters;

string s = New\_line\_without\_spaces(str);

vector<pair<TLexeme, int>> v;

int i = 0;

pair<bool, int> temp = Check\_unary\_minus(s, i);

if (temp.first)

{

if (temp.second == 0)

{

TLexeme p = Check\_number(s, i, positive);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 1)

{

TLexeme p = Check\_number(s, i, negative);

v.push\_back(make\_pair(p, i - 1));

}

else if (temp.second == 2 || temp.second == 5)

{

TLexeme p('-', unary\_operation);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 3)

{

TLexeme p0('-', unary\_operation);

v.push\_back(make\_pair(p0, i - 1));

TLexeme p('(');

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 4)

{

TLexeme p('(');

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 6)

{

TLexeme p0('-', unary\_operation);

v.push\_back(make\_pair(p0, i - 1));

parameters[s[i]].push(v.size());

TLexeme p(0.0);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 7)

{

parameters[s[i]].push(v.size());

TLexeme p(0.0);

v.push\_back(make\_pair(p, i));

}

i++;

}

for (; i < s.size(); i++)

{

int k = i;

temp = Check\_unary\_minus(s, i);

if (temp.first)

{

if (temp.second == 0)

{

if (s[k - 1] != '+')

{

TLexeme p0('+');

v.push\_back(make\_pair(p0, i - 1));

}

TLexeme p = Check\_number(s, i, positive);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 1)

{

TLexeme p = Check\_number(s, i, negative);

v.push\_back(make\_pair(p, i - 1));

}

else if (temp.second == 2 || temp.second == 5)

{

TLexeme p('-', unary\_operation);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 3)

{

TLexeme p0('-', unary\_operation);

v.push\_back(make\_pair(p0, i - 1));

TLexeme p('(');

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 4)

{

if (s[k - 1] != '+')

{

TLexeme p0('+');

v.push\_back(make\_pair(p0, i - 1));

}

TLexeme p('(');

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 6)

{

TLexeme p0('-', unary\_operation);

v.push\_back(make\_pair(p0, i - 1));

parameters[s[i]].push(v.size());

TLexeme p(0.0);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 7)

{

if (s[k - 1] != '+')

{

TLexeme p0('+');

v.push\_back(make\_pair(p0, i - 1));

}

parameters[s[i]].push(v.size());

TLexeme p(0.0);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 8)

{

TLexeme p0('-');

v.push\_back(make\_pair(p0, i - 1));

TLexeme p = Check\_number(s, i, positive);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 9)

{

TLexeme p0('-');

v.push\_back(make\_pair(p0, i - 1));

TLexeme p('(');

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 10)

{

TLexeme p0('-');

v.push\_back(make\_pair(p0, i - 1));

parameters[s[i]].push(v.size());

TLexeme p(0.0);

v.push\_back(make\_pair(p, i));

}

else if (temp.second == 11)

{

TLexeme p('-');

v.push\_back(make\_pair(p, i));

}

}

else if (s[i] == '-' || s[i] == '+' || s[i] == '\*' || s[i] == '/' || s[i] == '(' || s[i] == ')')

{

TLexeme p(s[i]);

v.push\_back(make\_pair(p, i));

}

else if ((s[i] >= '0' && s[i] <= '9') || s[i] == '.')

{

TLexeme p = Check\_number(s, i, positive);

v.push\_back(make\_pair(p, i));

}

else if (s[i] >= 'a' && s[i] <= 'z')

{

parameters[s[i]].push(v.size());

TLexeme p(0.0);

v.push\_back(make\_pair(p, i));

}

else

{

pair<int, int> err(unknown\_symbol, i);

throw err;

}

}

Set\_parameters(parameters, v, input);

return v;

}

Перевод в обратную польскую запись

vector<pair<TLexeme, int>> Create\_RPN\_array(const vector<pair<TLexeme, int>>& v)

{

map<pair<char, int>, int> m;

m[make\_pair('-', unary\_operation)] = 3;

m[make\_pair('\*', binary\_operation)] = 2;

m[make\_pair('/', binary\_operation)] = 2;

m[make\_pair('+', binary\_operation)] = 1;

m[make\_pair('-', binary\_operation)] = 1;

m[make\_pair('(', op\_bracket)] = 0;

TStack<pair<TLexeme, int>> s;

vector<pair<TLexeme, int>> ans;

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

{

if (v[i].first.GetType() == number)

{

ans.push\_back(v[i]);

}

else if (v[i].first.GetValue().oper == '\*' || v[i].first.GetValue().oper == '/' || v[i].first.GetValue().oper == '+' || v[i].first.GetValue().oper == '-')

{

int priority = m[make\_pair(v[i].first.GetValue().oper, v[i].first.GetType())];

if (s.isEmpty() || m[make\_pair(s.front().first.GetValue().oper, s.front().first.GetType())] < priority)

{

s.push(v[i]);

}

else

{

while (!s.isEmpty() && m[make\_pair(s.front().first.GetValue().oper, s.front().first.GetType())] >= priority)

{

ans.push\_back(s.pop());

}

s.push(v[i]);

}

}

else if (v[i].first.GetType() == op\_bracket)

{

s.push(v[i]);

}

else if (v[i].first.GetType() == cl\_bracket)

{

while (!s.isEmpty() && s.front().first.GetType() != op\_bracket)

{

ans.push\_back(s.pop());

}

s.pop();

}

}

while (!s.isEmpty())

{

ans.push\_back(s.pop());

}

return ans;

}

Проверка того, может ли лексема заданного типа идти после исходной лексемы

bool Type\_checking(const vector<int>& v, int type)

{

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

{

if (v[i] == type)

{

return 1;

}

}

return 0;

}

Проверка на корректность расположения операндов и операций

void Error\_checking(const vector<pair<TLexeme, int>>& v)

{

map<int, vector<int>> m;

vector<int> a1 = { binary\_operation, cl\_bracket };

m[number] = a1;

vector<int> a2 = { op\_bracket, number, unary\_operation };

m[op\_bracket] = a2;

vector<int> a3 = { cl\_bracket, binary\_operation };

m[cl\_bracket] = a3;

vector<int> a4 = { op\_bracket, number, unary\_operation };

m[unary\_operation] = a4;

vector<int> a5 = { op\_bracket, number, unary\_operation };

m[binary\_operation] = a5;

TStack<pair<TLexeme, int>> s;

if (v[0].first.GetType() == cl\_bracket || v[0].first.GetType() == binary\_operation)

{

pair<int, int> err(incorrect\_first\_symbol, v[0].second);

throw err;

}

else if (v[v.size() - 1].first.GetType() == unary\_operation || v[v.size() - 1].first.GetType() == binary\_operation || v[v.size() - 1].first.GetType() == op\_bracket)

{

pair<int, int> err(incorrect\_last\_symbol, v[v.size() - 1].second);

throw err;

}

for (int i = 0; i < v.size() - 1; i++)

{

if (v[i].first.GetType() == op\_bracket)

{

s.push(v[i]);

}

else if (v[i].first.GetType() == cl\_bracket)

{

if (!s.isEmpty() && s.front().first.GetType() == op\_bracket)

{

s.pop();

}

else

{

pair<int, int> err(wrong\_bracket\_sequence, v[i].second);

throw err;

}

}

vector<int> t = m[v[i].first.GetType()];

if (Type\_checking(t, v[i + 1].first.GetType()))

{

continue;

}

else

{

pair<int, int> err(missed\_operation\_or\_operand, v[i].second);

throw err;

}

}

if (v[v.size() - 1].first.GetType() == op\_bracket)

{

s.push(v[v.size() - 1]);

}

else if (v[v.size() - 1].first.GetType() == cl\_bracket)

{

if (!s.isEmpty() && s.front().first.GetType() == op\_bracket)

{

s.pop();

}

else

{

pair<int, int> err(wrong\_bracket\_sequence, v[v.size() - 1].second);

throw err;

}

}

if (!s.isEmpty())

{

pair<int, int> err(wrong\_bracket\_sequence, v[v.size() - 1].second);

throw err;

}

}

Вычисление значения выражения

double Calculate(const string& s, const vector<double>& input)

{

vector<pair<TLexeme, int>> v = Create\_lexeme\_array(s, input);

Error\_checking(v);

double ans = Solver(Create\_RPN\_array(v));

return ans;

}

Обработка ошибок

string Error\_output(pair<int, int> err)

{

string temp;

switch (err.first)

{

case division\_by\_zero:

temp = "Division by zero in symbol: ";

break;

case incorrect\_point:

temp = "Incorrect point in symbol: ";

break;

case unknown\_symbol:

temp = "Unknown symbol: ";

break;

case incorrect\_first\_symbol:

temp = "Incorrect first symbol: ";

break;

case incorrect\_last\_symbol:

temp = "Incorrect last symbol: ";

break;

case wrong\_bracket\_sequence:

temp = "Wrong bracket sequence: ";

break;

case missed\_operation\_or\_operand:

temp = "Missed operration or opperand: ";

break;

default:

temp = "Unknown error: ";

}

return temp;

}