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

Митин Р.А.

**Проверил**:

преподаватель каф. МОСТ,

Волокитин В.Д.

Нижний Новгород

2021

**Содержание**

[Постановка задачи 3](#_Toc26962562)

[Метод решения 4](#_Toc26962563)

[Руководство пользователя 5](#_Toc26962564)

[Описание программной реализации 6](#_Toc26962565)

[Подтверждение корректности 7](#_Toc26962566)

[Результаты экспериментов 8](#_Toc26962567)

[Заключение 9](#_Toc26962568)

[Приложение 10](#_Toc26962569)

# Постановка задачи

Разработать программу, выполняющую вычисление арифметического выражения с вещественными числами. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимые операции известны: +, -, /, \*. Допускается наличие знака "-" в начале выражения или после открывающей скобки. Программа должна выполнять предварительную проверку корректности выражения и сообщать пользователю вид ошибки и номера символов строки, в которых были найдены ошибки.

# Метод решения

Вычисление выражение происходит в несколько этапов. На первом этапе проверяется корректность введённого пользователем арифметического выражения. С помощью цикла по входным данным и множественным условным операторам, проверяется корректность, в случае несоответствия правилам ввода бросается исключение, которое обрабатывается в основной программе. После ввода пользователем корректного выражения происходит преобразования входной строки в токены. Каждый токен имеет один из 5 типов: переменная, число, операция, скобка, функция. Преобразование происходит с помощью сравнения очередного входного символа с некоторыми другими наборами символов. Каждый входной символ принадлежит одному из наборов, поскольку введённое выражение корректное, при этом наборы не имеют общих символов(за исключением минуса, для его обработки программ проверяет между какими символами он находится и в зависимости от этого определяет унарный это минус или операция) , соответственно все символы можно верно обработать и определить их тип. Результат записывается в вектор, далее вектор с помощью функции преобразуется из инфиксной формы записи в постфиксную. Алгоритм преобразования следующий:

1. При встрече операнда помещаем его в выходной вектор
2. Если встретилась (, то кладём её в стек
3. Если встретилась ), то вынимаем из стека элементы и записываем их в выходной вектор до тех пор, пока не встретиться (, при встрече просто вынимаем её из стека и ни куда не записываем
4. Если встретилась операция, то выталкиваем из стека все операции приоритет которых не ниже данной и записываем их в выходную строку, после чего записываем операцию в стек.
5. При достижении конца входного вектора выгружаем все элементы из стека в выходной вектор.

Далее программе необходимо получить значения переменных, для этого создаётся массив указателей на все переменные и на каждой итерации цикла проверяется присвоены значения всем переменным или нет, если нет то пользователя просят ввести значения переменной в формате x=3, после чего программа всем переменным с именем x присваивает значение 3.

На последнем этапе необходимо произвести вычисления выражения в постфиксной записи, для этого действуем по следующему алгоритму:

1. При встрече операнда кладём его в стек
2. При встречи операции достаём из стека два операнда, применяем операцию и кладём обратно в стек полученный результат
3. При встрече функции, достаём из стека последний элемент, применяем к нему функцию и кладём обратно результат

После выполнения данного алгоритма будет получен стек с одним элементом, который и будет является значением выражения

# Руководство пользователя

При старте программы пользователь должен ввести арифметическое выражение без пробелов. В качестве операций поддерживается +,-,\*,/, а так же унарный минус. В качестве операндов можно использовать числа и переменные. Имена переменных представляет собой одну букву латинского алфавита. Строчные и заглавные буквы являются разными переменными. Можно использовать скобки для задания приоритета операций. При вводе неверного выражения будет выведено описание ошибки и индекс символа с ошибкой. При вводе верного выражения программа выведет полученный результат и запросит значения переменных, ввод происходит с следующем формате <Имя переменной> = <Значение>, допускается использовать произвольное количество пробелов и знаков равно (при желании их можно опустить). После ввода программа преобразует выражение в постфиксную форму записи, выведет её, а затем произведёт вычисление и выведет результат на экран.

# Описание программной реализации

В решении содержаться 4 проекта: arithmetic, gtest, sample, tests

Проект arithmetic содержит в себе файлы arithmetic.h, arithmetic.cpp, stack.h, vector.h. Последние 2 файла содержат в себе реализацию соответствующих структур данных. В arithmetic.h содержаться объявления классов и функций необходимых для работы с арифметическими выражениями, в arithmetic.cpp их реализация.

Проект gtest необходим для возможности тестирования функций программы

Проект sample содержит в себе единственный файл main\_ arithmetic, в котором реализовано пользовательское приложение

В проекте tests содержаться 3 файла test\_main.cpp, test\_stack.cpp, test\_arithmetic.cpp. В последних двух представлен набор тестов для соответствующих классов и функций, файл test\_main.cpp необходим для запуска тестов.

# Подтверждение корректности

Для подтверждения корректности в программе используется система тестирования google test, все нетривиальные функции работающие не с консолью проверяются тестами на корректность.

# Заключение

Была разработана программа, выполняющая вычисление арифметического выражения с вещественными числами. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимые операции: +, -, /, \*. Допускается наличие знака "-" в начале выражения или после открывающей скобки. Программа должна выполняет предварительную проверку корректности выражения и сообщать пользователю вид ошибки и индексы символов строки, в которых были найдены ошибки.

# Приложение

bool correct(std::string input)

{

int count\_brack = 0;

int open\_brack\_ind = -1;

bool is\_open = false;

bool dot = false;

if (input.empty())

return true;

if (input[0] == ')' || operation() == input[0] && input[0] != '-')

throw exception("Wrong beggining of the expretion", 0);

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

{

if (variable() != input[i] && number() != input[i] && operation() != input[i] && bracket() != input[i])

throw exception("Wrong symbol", i);

if (input[i] == '.' && !dot)

{

dot = true;

}

else

{

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

throw exception("Two dots in number", i);

}

if (number() != input[i])

dot = false;

if (bracket() == input[i])

{

if (input[i] == '(')

{

if (!count\_brack)

open\_brack\_ind = i;

count\_brack++;

is\_open = true;

}

else

{

if (count\_brack == 0)

throw exception("The opening parenthesis is missing", i);

count\_brack--;

is\_open = false;

}

}

if (i != 0 && variable() == input[i] && variable() == input[i - 1])

throw exception("Name for variable is too long, use one symbol for variable", i);

if (i != 0 && bracket() == input[i] && (number() == input[i - 1] && variable() == input[i - 1]))

throw exception("Bracket after number or variable", i);

if (i != 0 && bracket() == input[i-1] && (number() == input[i] && variable() == input[i]))

throw exception("Number or variable after bracket", i);

if (i != 0 && (number() == input[i - 1] || variable() == input[i - 1]) && \

operation() != input[i] && input[i] != ')' && number() != input[i])

throw exception("Number or variable before bracket", i);

if (i != 0 && (number() == input[i] || variable() == input[i]) && \

operation() != input[i - 1] && input[i - 1] != '(' && number() != input[i - 1])

throw exception("Number or variable after bracket", i);

if (i != 0 && '-' == input[i - 1] && number() != input[i] && variable() != input[i] && bracket() != input[i] && '-' != input[i])

throw exception("Not a number or bracket after -", i);

if (i != 0 && operation() == input[i - 1] && (operation() == input[i] && '-' != input[i]))

throw exception("Incorrect symbol after operation", i);

if (i != 0 && input[i - 1] == '.' && (number() != input[i] || input[i] == '.'))

throw exception("Number cant end with dot", i);

}

if (input[input.size() - 1] == '(' || operation() == input[input.size() - 1] || input[input.size() - 1] == '.')

throw exception("Wrong ending of the expretion", input.size() - 1);

if (count\_brack)

throw exception("The closing bracket is missing", open\_brack\_ind);

return true;

}

token getnumber(std::string s, int& index)

{

double result = 0.0;

while (index < s.size() && s[index] >= '0' && s[index] <= '9')

{

result \*= 10.0;

result += s[index] - '0';

index++;

}

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

{

index++;

double power = 0.1;

while (index < s.size() && s[index] >= '0' && s[index] <= '9')

{

result += (s[index] - '0') \* power;

power \*= 0.1;

index++;

}

}

token tmp;

tmp.type = Number;

tmp.num = number(result);

return tmp;

}

vector<token> parcer(std::string input)

{

vector<token> result;

int index = 0;

while (index != input.size())

{

if (number() == input[index])

result.push\_back(getnumber(input, index));

if (index == input.size())

break;

if (variable() == input[index])

{

token tmp;

tmp.var = variable(input[index++]);

tmp.type = Variable;

result.push\_back(tmp);

}

if (index == input.size())

break;

if (bracket() == input[index])

{

token tmp;

tmp.brack = bracket(input[index++] == '(');

tmp.type = Bracket;

result.push\_back(tmp);

}

if (index == input.size())

break;

if (operation() == input[index])

{

token tmp;

if (input[index] != '-')

{

tmp.type = Operation;

tmp.oper = operation(input[index]);

}

else

{

if (index == 0 || input[index - 1] == '(' || operation() == input[index - 1])

{

tmp.type = Function;

}

else

{

tmp.type = Operation;

tmp.oper = operation(input[index]);

}

}

result.push\_back(tmp);

index++;

}

}

return result;

}

vector<token> to\_polish(vector<token> vec)

{

vector<token> result;

stack<token> st;

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

{

switch (vec[i].type)

{

case Variable:

case Number:

result.push\_back(vec[i]);

break;

case Operation:

while (!st.is\_empty() && st.top().type != Bracket\

&& oper\_priority(vec[i].oper) >= oper\_priority(st.top().oper))

result.push\_back(st.pop());

st.push\_back(vec[i]);

break;

case Bracket:

if (vec[i].brack.open)

{

st.push\_back(vec[i]);

}

else

{

while (st.top().type != Bracket)

result.push\_back(st.pop());

st.pop();

}

break;

case Function:

st.push\_back(vec[i]);

}

}

while (!st.is\_empty())

{

result.push\_back(st.pop());

}

return result;

}

double compute(vector<token> expr)

{

if (expr.empty())

return 0.0;

stack<double> result;

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

{

switch (expr[i].type)

{

case Variable:

result.push\_back(expr[i].var.value);

break;

case Number:

result.push\_back(expr[i].num.val);

break;

case Operation:

{

double tmp = result.pop();

tmp = expr[i].oper.operator()(result.pop(), tmp);

result.push\_back(tmp);

break;

}

case Function:

{

double tmp = result.pop();

tmp \*= -1;

result.push\_back(tmp);

break;

}

default:

break;

}

}

return result.pop();

}