Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт Информационных технологий, Математики и Механики

Отчёт по лабораторной работе

Вычисление арифметических выражений (стеки)

Выполнил:

студент ф-та ИТММ гр. 381808-2

Осипов Николай

Проверил:

ассистент каф. МО ЭВМ, ВМК

Панов А

Нижний Новгород

2019 г.

**Содержание**

Введение........................................................................................................................................3

Постановка задачи........................................................................................................................4

Руководство пользователя...........................................................................................................5

Руководство программиста.........................................................................................................6

Описание структур данных.........................................................................................................7

Заключение...................................................................................................................................9 Литература....................................................................................................................................10Приложения.................................................................................................................................11

Исходный код программы..........................................................................................................11

**Введение**

При вычислении произвольных арифметических выражений возникают две основные задачи: проверка корректности введённого выражения и выполнение операций в порядке, определяемом их приоритетами и расстановкой скобок. Существует алгоритм, позволяющий реализовать вычисление произвольного арифметического выражения за один просмотр без хранения промежуточных результатов. Для реализации данного алгоритма выражение должно быть представлено в постфиксной форме. Рассматриваемые в данной лабораторной работе алгоритмы являются начальным введением в область машинных вычислений.

**Постановка задачи**

В рамках лабораторной работы ставится задача разработки программных средств, производящих обработку арифметических выражений, включая проверку правильности записи выражения, перевод в постфиксную форму и вычисление результата.

**Руководство пользователя**

По результатам работы была реализована структура lexeme для того, чтобы ею пользоваться необходимо:

1. Создать новый проект.
2. Добавить существующие элементы и выбрать Postfix.h и lexeme.cpp.
3. Назначить наш проект автозагружаемым.
4. Теперь нашему проекту доступна структура lexeme.

**Руководство программиста**

**Описание структуры программы**

Составляющие проекта LAB3-POSTFIX:

* LAB3-POSTFIX – приложение, вычисляющее арифметические выражения;
* Postfix.h – заголовочный файл класса TPostfix;
* lexeme.cpp – исполняемый файл класса TPostfix;

**Описание структуры данных**

В данной лабораторной работе, мы получаем на вход арифметическое выражение в инфиксной форме.

Наша задача перевести арифметическое выражение из инфиксной формы записи в постфиксную и подсчитать результат.

Отличительной особенностью обратной польской нотации является то, что:

* Все аргументы (или операнды) расположены перед знаком операции.
* Запись набора операций состоит из последовательности операндов и знаков операций. Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.
* Результатом вычисления выражения становится результат последней вычисленной операции.

Алгоритм перевода из инфиксной формы записи в постфиксную:

1. Преобразуем входное выражение в вектор из лексем с помощью метода Parse.
   1. В ней используется метод getNextLexeme, в нем мы рассматриваем каждый символ из входной строки и выводим лексему в метод Parse.
   2. В методе Parse, учитывая статус алгоритма (начало, ожидание числа, ожидание операнда), определяем тип полученной лексемы (число, сумма, минус, умножение, деление, степень, скобка), меняем статус алгоритма на основе типа лексемы и вставляем лексему в вектор, который выводит метод Parse
2. С помощью метода VectorToStack преобразуем вектор лексем из метода Parse в вектор лексем в постфиксной форме. Алгоритм добавляет в вектор лексему – числа, добавляет в стек арифметические операции и скобки, в случае наличия этих лексем в стеке и если приоритет обрабатываемой операнды выше, чем у операнды на вершине стека, начинает добавлять в вектор операцию на вершине стека, затем удаляет эту вершину и добавляет в стек обрабатываемую операцию. Если алгоритм встретит закрывающую скобку, то начнет добавлять в вектор вершину стека, затем удалять эту вершину до того момента, пока не встретит в стеке открывающую скобку. После удалит эту открывающую скобку. В конце, когда алгоритм обработает каждую лексему из исходного вектора, он будет добавлять вершину стека в вектор и удалять вершину, пока стек не окажется пустым. В итоге метод VectorToStack возвращает вектор.
3. В функцию PolishCalculate поступает вектор из метода VectorToStack. В ней алгоритм записывает лексемы – числа из вектора в стек. Если алгоритм находит лексему-операцию, то создает необходимые лексемы, в которые записывает верхние числа из стека, удаляет эти головные числа из стека и добавляет в стек значение, которое он получил после определенной операции (на основе типа лексемы - операции) между значениями лексем-чисел. В итоге функция возвращает значение единственной оставшейся лексемы.

Описание алгоритмов

Структура lexeme

* vector<lexeme> Parse(string s)

преобразует входную строку в вектор лексем

* lexeme getNextLexeme(string s, int pos, int &nexPos)

создает лексемы в методе Parse из символов входной строки

* vector<lexeme> VectorToStack(vector<lexeme> data)

возвращает вектор лексем в порядке постфиксной записи

* double PolishCalculate(vector<lexeme> data)

принимает вектор лексем в постфиксной форме и возвращает ответ

**Заключение**

Целью работы была реализация структуры данных, вычисляющая значение арифметических выражений.

В результате была написана структура данных, которая проверяет введённое арифметическое выражение, переводит арифметическое выражение из инфиксной формы записи в постфиксную и подсчитывает результат.

Программные средства созданы при помощи среды программирования С++ и содержат:

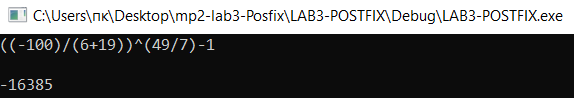
* структуру lexeme

**Литература**

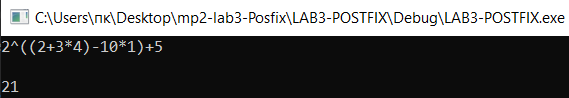
Лабораторный практикум. Составители: Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017. – 105с

**Приложения**

Приложение 1



Приложение 2



## **Исходный код**

Lexeme.

bool TPostfix::IsOperator(char v)

{

if (v == '~' || v == '-' || v == '+' || v == '/' || v == '\*' || v == '^' || v == '(' || v == ')')

return true;

else

return false;

}

bool TPostfix::IsNumeral(char v)

{

if (int(v) >= 48 && int(v) <= 57)

return true;

return false;

}

bool TPostfix::CheckOperations()

{

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

if(!IsNumeral(infix[i]) && !IsOperator(infix[i]))

return false;

int Operands = 0, Operations = 0;

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

{

if (infix[i] == '(' || infix[i] == ')')

{

i++;

if (i == infix.size())

break;

}

if (infix[i] == '-' || infix[i] == '+' || infix[i] == '/' || infix[i] == '\*' || infix[i] == '^')

Operations++;

if (!IsOperator(infix[i]))

{

while (!IsOperator(infix[i]))

{

i++;

if (i == infix.size())

break;

}

i--;

Operands++;

}

}

if (Operands == Operations + 1)

return true;

else return false;

}

bool TPostfix::CheckBrackets()

{

int LeftBrackets = 0, RightBrackets = 0;

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

{

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

LeftBrackets++;

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

RightBrackets++;

if (RightBrackets > LeftBrackets)

throw "Error Brackets";

}

if (LeftBrackets == RightBrackets)

return true;

else return false;

}

int TPostfix::Priority(char a, char b)

{

if (a == '~')

return 1;

if (b == '~')

return -1;

if (a == '^' && b != '~')

return 1;

if (b == '^' && a != '~')

return -1;

if ((a == '\*' || a == '/') && (b == '\*' || b == '/') || (a == '+' || a == '-') && (b == '+' || b == '-'))

return 0;

if ((a == '\*' || a == '/') && (b == '+' || b == '-'))

return 1;

if ((a == '+' || a == '-') && (b == '\*' || b == '/'))

return -1;

throw "not operation";

}

void TPostfix::SplitStr(string& str)

{

if (str.empty())

return;

if (str.back() != ' ')

str += ' ';

}

string TPostfix::ToPostfix()

{

if (!CheckBrackets())

throw "Error Brackets";

for (int j = 0; j < infix.size(); j++)

{

if (infix[0] == '-')

infix[0] = '~';

if (infix[j] == '-' && (infix[j-1] == '(' || infix[j - 1] == '~'))

infix[j] = '~';

}

int count = -1;

stack<char> Oper;

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

{

if (IsOperator(infix[i]))

{

if ((Oper.empty() || infix[i] == '(') && infix[i]!='~')

{

if (infix[i] != '(')

SplitStr(postfix);

Oper.push(infix[i]);

count++;

continue;

}

else if ((Oper.empty() || Oper.top() == '~') && infix[i] == '~')

{

Oper.push(infix[i]);

count++;

continue;

}

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

{

while (Oper.top() != '(')

{

SplitStr(postfix);

postfix += Oper.top();

Oper.pop();

count--;

}

Oper.pop();

count--;

}

else if ((Oper.top()) == '(' || Priority(infix[i], Oper.top()) == 1)

{

Oper.push(infix[i]);

SplitStr(postfix);

count++;

}

else if (Priority(infix[i], Oper.top()) == -1)

{

SplitStr(postfix);

while (!(Oper.empty()))

{

if (Oper.top() == '(')

break;

postfix += Oper.top();

Oper.pop();

SplitStr(postfix);

count--;

}

Oper.push(infix[i]);

count++;

}

else if (Priority(infix[i], Oper.top()) == 0)

{

SplitStr(postfix);

postfix += Oper.top();

Oper.pop();

Oper.push(infix[i]);

SplitStr(postfix);

}

}

else

postfix += infix[i];

if (i == infix.size() - 1)

{

for (count; count >= 0; count--)

{

SplitStr(postfix);

postfix += Oper.top();

Oper.pop();

}

}

}

return postfix;

}

double TPostfix::Calculate()

{

if (!CheckOperations() || postfix == "")

throw "Error Operand";

stack<double> Res;

string str;

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

{

if (postfix[i] == ' ')

continue;

if (!IsOperator(postfix[i]))

{

while (postfix[i] != ' ')

{

str += postfix[i];

i++;

}

Res.push(atof(str.c\_str()));

str = "";

}

else

{

if (postfix[i] == '~')

{

double a = Res.top();

Res.pop();

Res.push(-a);

continue;

}

double a = Res.top();

Res.pop();

double b = Res.top();

Res.pop();

if (postfix[i] == '+')

Res.push(b + a);

else if (postfix[i] == '-')

Res.push(b - a);

else if (postfix[i] == '\*')

Res.push(b \* a);

else if (postfix[i] == '/')

{

if (a == 0.0)

throw "Error Division";

Res.push(b / a);

}

else if (postfix[i] == '^')

Res.push(pow(b,a));

}

}

return Res.top();

}