Федеральное государственное автономное образовательное учреждение

высшего образования

«Нижегородский государственный университет

им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Вычисление арифметических выражений

Выполнил:

студент ИИТММ ПМИ гр. 381703-1

Лаптев В. В.

Проверил:

Ассистент кафедры МОСТ ИИТММ

Волокитин В. Д.

Нижний Новгород

2018 г.

Содержание

Введение3

Постановка задачи4

Руководство пользователя5

Руководство программиста6

Описание структур данных6

Описание алгоритмов8

Описание компонентов программного комплекса.11

Заключение.12

Список литературы.13

Приложение14

**Введение**

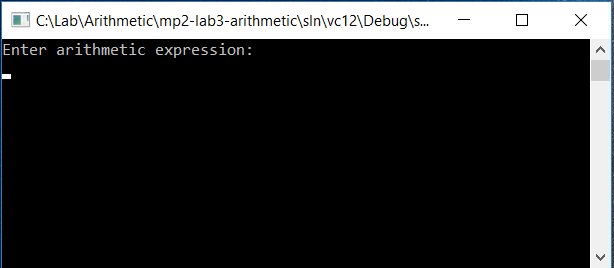
В различных практических задачах люди регулярно сталкиваются с вычислениями. Бывает так, что выражение, которое нужно вычислить получается весьма громоздким, а его подсчёт “вручную” занимает много времени. Именно поэтому возникает необходимость разработки программы, которая умеет работать с арифметическими выражениями и способна быстро их вычислять.

**Постановка задачи**

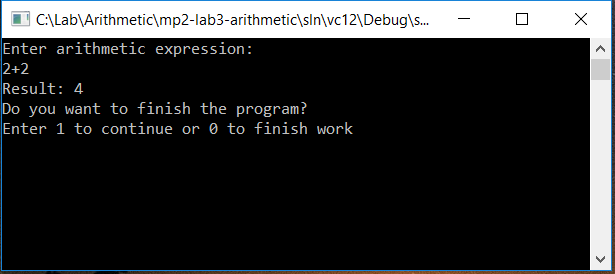
Требуется разработать программу, выполняющую вычисление арифметического выражения с вещественными числами. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимые операции известны: +, -, /, \*. Допускается наличие знака "-" в начале выражения или после открывающей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

**Руководство пользователя**

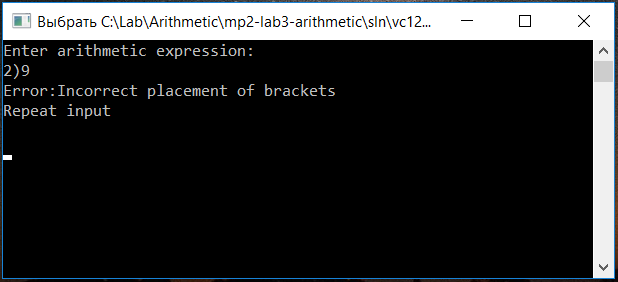
Для начала работы программы нужно запустить на выполнение файл arithmetic.exe. В появившемся окне необходимо ввести арифметическое выражение (без пробелов).



Если введённое выражение корректно, то программа выводит результат на экран.



В противном случае, будет выведено сообщение об ошибке и пользователю будет предложено повторить ввод, до тех пор, пока выражение не станет корректным.



После вывода результата пользователю предлагается продолжить или завершить программу путём ввода в консоль 1 или 0.

**Руководство программиста**

**Описание структур данных**

1. *class Lexeme* - структура данных, предназначенная для разбиения на части, приходящего в программу арифметического выражения в виде строки. Это упрощает написание кода и позволяет повысить его читаемость.

* *string Name*

в случае если тип лексемы – число или переменная, то поле хранит название переменной, иначе в поле содержится информация о том какую операцию или скобку определяет данная лексема.

* *TTag Type*

поле, задающее один из четырёх типов лексемы: NUMBER, OPERATION, BRACKET, VARIABLE

* *double Value*

поле хранит числовое значение если текущая лексема число или переменная, в противном случае не используется.

class Lexeme

{

TTag Type; //тип лексемы

double Value; //значение лексемы

string Name; //"имя" лексемы

public:

Lexeme(); //конструктор по умолчанию

Lexeme(TTag, double); //конструктор с параметрами

Lexeme(TTag, string, double = 0.0); //конструктор с параметрами

~Lexeme(); //деструктор

TTag GetType(); //получить тип

double GetValue(); //получить значение

string GetName(); //получить "имя"

void SetValue(double); //установить значение

bool operator==(const Lexeme&)const; //сравнение

bool operator!=(const Lexeme&)const; //сравнение

};

1. *class TStack* – структура предназначенная для хранения данных. В данной лабораторной работе структура TStack используется для реализации алгоритмов вычисления арифметических выражений.

template< class ValType>

class TStack

{

int top; //вершина

int size; //размер

ValType \*stk;

void Realoc();//перевыделение памяти под стек

public:

TStack(); //конструктор по умолчанию

TStack(int s); //конструктор с параметром

~TStack(); //деструктор

void Push(ValType elem); //положить элемент в стек

ValType Pop(); //достать элемент из стека

ValType CheckTop(); //посмотреть кто на вершине стека

bool IsEmpty(); //проверка на пустоту

int GetSize(); //получить размер

void Clean(); //очистить стек

};

**Описание алгоритмов**

1. *Алгоритм перевода строки в массив лексем (реализацию алгоритма см. в Приложении)*

Вход: строка

1. Создаём вектор лексем для результата
2. Пока не конец строки проверяем символ:
3. Если символ – буква, то копим последовательность букв, затем кладём в вектор
4. Если символ – число, то копим число, затем кладём в вектор
5. Если символ – операция, кладём в вектор
6. Если символ – скобка, кладём в вектор
7. Если встретили неразрешённый символ, то выбрасываем исключение.

Выход: вектор лексем

2. *Алгоритм проверки корректности выражения (реализацию алгоритма см. в Приложении)*

Вход: вектор лексем

1. Создаём стек для проверки расстановки скобок
2. Пока не конец вектора:
3. Если символ – открывающая скобка, то кладём в стек
4. Если символ – закрывающая скобка
5. Если стек пуст, то выбрасываем исключение
6. Иначе извлекаем из стека элемент
7. Проверяем: Если стек не пуст, то выбрасываем исключение
8. Проверяем первый элемент вектора на корректность, если лексема некорректна, выбрасываем исключение
9. Проверяем последний элемент вектора на корректность, если лексема некорректна, выбрасываем исключение
10. Пока не конец вектора:
11. Если за текущей лексемой стоит некорректная лексема, то выбрасываем исключение

Выход: логическое значение true либо false

3. *Алгоритм перевода массива лексем в обратную польскую запись* *(реализацию алгоритма см. в Приложении)*

Вход: массив лексем

1. Создаём стек, в который будем складывать операции и скобки
2. Создаём результирующий вектор
3. Пока не конец массива лексем:
4. Если текущая лексема – бинарная операция, то
5. Если стек пустой или приоритет текущей операции больше приоритета той, что в стеке, то кладём операцию в стек
6. Иначе:
7. Пока приоритет текущей операции меньше либо равен приоритета операции на вершине стека и стек не пуст будем извлекать элемент из стека и складывать их в результирующий вектор
8. После завершения цикла необходимо положить текущую операцию в стек
9. Если текущая лексема – унарная операция, то кладём её в стек
10. Если текущая лексема - открывающая скобка, то кладём её в стек
11. Если текущая лексема - закрывающая скобка, то извлекаем всех из стека в результирующий вектор, пока не дойдём до открывающей скобки
12. Если текущая лексема - число или переменная, то кладём лексему в результирующий вектор
13. Извлекаем всех, кто остался в стеке в результирующий вектор

Выход: вектор лексем в обратной польской записи

4. *Алгоритм вычисления арифметического выражения (реализацию алгоритма см. в Приложении)*

Вход: вектор лексем в обратной польской записи

1. Создаём стек, в который будем складывать результаты вычисления операций
2. Пока не конец вектора:
3. Если текущая лексема не операция положить в стек
4. Иначе:
5. Если текущая операция бинарная, то извлекаем из стека два операнда и проводим над ними текущую операцию, результат кладём в стек
6. Если текущая операция унарная, то извлекаем один операнд и выполняем над ним указанную операцию, после чего результат кладём в стек
7. Результат вычисления лежит на вершине стека

Выход: число типа double

5. *Алгоритм установки значений для переменных (реализацию алгоритма см. в Приложении)*

Вход: ссылка на вектор лексем

1. Создаём вектор лексем для того, чтобы помнить переменные, которым мы присвоим значения.
2. Пока не конец вектора:
3. Если встретили переменную, то
4. Если она нашлась в вспомогательном векторе, то переходим к следующей итерации
5. Иначе: просим пользователя задать значение переменной и записываем ее в вспомогательный вектор

Выход: нет возвращаемого значения

**Описание структур программного комплекса**

Программа состоит из следующих основных файлов:

1. В файле arithmetic.h представлен интерфейс класса Lexeme и прототипы функций;
2. В файле arithmetic.cpp реализованы методы класса Lexeme и функции необходимые для вычисления арифметического выражения;
3. В файле stack.h представлен интерфейс класса Stack и реализованы его методы;
4. В файле main\_arithmetic.cpp находится основная программа.
5. Файл test\_stack.cpp содержит тесты для класса Stack.
6. Файл test\_arithmetic.cpp содержит тесты для класса Lexeme и функций, которые проверяют корректность выражения, переводят его в обратную польскую запись и вычисляют значение.
7. Файл test\_main.cpp запускает все google tests.

**Заключение**

В данной лабораторной работе удалось реализовать программный комплекс, способный вычислять различные арифметические выражения.

Достигнутые результаты:

1. Корректное вычисление арифметического выражения
2. Поддержка следующих бинарных операций: +,-,\*,/.
3. Поддержка унарного минуса
4. Поддержка переменных в выражении
5. Диалоговое пользовательское приложение
6. Реализация тестов для корректности работы программы

**Литература**

1. Алгоритм преобразования выражения в ОПЗ и его последующего вычисления [http://trubetskoy1.narod.ru/ppn.html]
2. Т. Кормен, Ч. Лейзерсон, Р. Ривест Алгоритмы: построение и анализ. М.: МЦНМО, 1999.-960 с., 263 ил.

**Приложение**

**Реализация методов класса Lexeme:**

1. *Перевод строки в массив лексем*

vector<Lexeme> Convert\_to\_lexeme\_array(string exp)

{

vector<Lexeme> v;

bool error = true;

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

{

if ( (exp[i] >= 'A' && exp[i] <= 'Z') || (exp[i] >= 'a' && exp[i] <= 'z') )

{

string name;

while ((exp[i] >= 'A' && exp[i] <= 'Z') || (exp[i] >= 'a' && exp[i] <= 'z'))

{

name += exp[i];

i++;

}

i--;

Lexeme temp(VARIABLE, name);

v.push\_back(temp);

error = false;

}

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

{

string integ, fract, res;

while (exp[i] >= '0' && exp[i] <= '9')

{

integ += exp[i];

i++;

}

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

{

i++;

while (exp[i] >= '0' && exp[i] <= '9')

{

fract+= exp[i];

i++;

}

i--;

res = integ + "." + fract;

}

else

{

i--;

res = integ;

}

Lexeme temp(NUMBER, atof(res.c\_str()));

v.push\_back(temp);

error = false;

}

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

{

if (!i || v.back().GetType() == OPERATION || v.back().GetName() == "(")

{

Lexeme temp(OPERATION, "unar-");

v.push\_back(temp);

error = false;

}

else

{

Lexeme temp(OPERATION, "-" );

v.push\_back(temp);

error = false;

}

}

if (exp[i] == '+' || exp[i] == '\*' || exp[i] == '/')

{

string str;

str += exp[i];

Lexeme temp(OPERATION, str);

v.push\_back(temp);

error = false;

}

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

{

string str;

str += exp[i];

Lexeme temp(BRACKET, str);

v.push\_back(temp);

error = false;

}

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

{

throw Error("Unresolved dot in expression");

}

if (error)

{

throw Error("Invalid symbol in expression");

}

error = true;

}

return(v);

}

1. *Проверка корректности выражения*

bool Checking\_correctness(vector<Lexeme> exp)

{

bool res = true;

Lexeme first = exp[0], last = exp[exp.size() - 1];

TStack<string> s;

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

{

if (exp[i].GetType() == BRACKET)

{

if (exp[i].GetName() == "(")

{

s.Push(exp[i].GetName());

}

else

{

if (s.IsEmpty())

{

throw Error("Incorrect placement of brackets");

}

else

{

s.Pop();

}

}

}

}

if (!s.IsEmpty())

{

throw Error("Incorrect placement of brackets");

}

if ((first.GetType() == OPERATION && first.GetName() != "unar-") || first.GetName() == ")")

{

throw Error("Unresolved lexeme at first");

}

if (last.GetType() == OPERATION || last.GetName() == "(")

{

throw Error("Unresolved lexeme at last");

}

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

{

if (exp[i].GetType() == NUMBER || exp[i].GetType() == VARIABLE)

{

if (exp[i + 1].GetType() == NUMBER || exp[i + 1].GetType() == VARIABLE || exp[i + 1].GetName() == "unar-" ||

exp[i + 1].GetName() == "(")

{

throw Error("Unresolved lexeme after number/variable");

}

}

if (exp[i].GetType() == OPERATION)

{

if ((exp[i + 1].GetType() == OPERATION && exp[i + 1].GetName() != "unar-") || exp[i + 1].GetName() == ")")

{

if (exp[i].GetName() == "unar-")

{

throw Error("Unresolved lexeme after unary operation.");

}

else

{

throw Error("Unresolved lexeme after binary operation.");

}

}

}

if (exp[i].GetName() == "(")

{

if ((exp[i + 1].GetType() == OPERATION && exp[i + 1].GetName() != "unar-") || exp[i + 1].GetName() == ")")

{

throw Error("Unresolved lexeme after opening bracket");

}

}

if (exp[i].GetName() == ")")

{

if (exp[i + 1].GetType() == NUMBER || exp[i + 1].GetType() == VARIABLE || exp[i + 1].GetName() == "unar-" ||

exp[i + 1].GetName() == "(")

{

throw Error("Unresolved lexeme after closing bracket");

}

}

}

return(res);

}

1. *Перевод в обратную польскую запись*

vector<Lexeme> Convert\_to\_postfix\_notation(vector<Lexeme> parsed)

{

TStack<Lexeme> s;

int size = 0;

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

{

if (parsed[i].GetType() != BRACKET)

size++;

}

vector<Lexeme> postf(size);

int i = 0, j = 0;

while(i < parsed.size())

{

if (parsed[i].GetType() == OPERATION)

{

if (s.IsEmpty() || Priority[parsed[i].GetName()] > Priority[s.CheckTop().GetName()])

{

s.Push(parsed[i]);

}

else

{

if (parsed[i].GetName() == "unar-")

{

s.Push(parsed[i]);

}

else

{

while (Priority[parsed[i].GetName()] <= Priority[s.CheckTop().GetName()])

{

postf[j] = s.Pop();

j++;

if (s.IsEmpty())

break;

}

s.Push(parsed[i]);

}

}

}

if (parsed[i].GetType() == BRACKET)

{

if (parsed[i].GetName() == "(")

{

s.Push(parsed[i]);

}

else

{

while (s.CheckTop().GetName() != "(")

{

postf[j] = s.Pop();

j++;

}

s.Pop();

}

}

if(parsed[i].GetType() == NUMBER || parsed[i].GetType() == VARIABLE)

{

postf[j] = parsed[i];

j++;

}

i++;

}

while (!s.IsEmpty())

{

postf[j] = s.Pop();

j++;

}

return(postf);

}

1. *Вычисление выражения в обратной польской записи*

double Expression\_result( vector<Lexeme> postf )

{

TStack<double> s;

int i = 0;

while (i < postf.size())

{

if (postf[i].GetType() != OPERATION)

s.Push(postf[i].GetValue());

else

{

double op = s.Pop();

if (postf[i].GetName() == "+")

s.Push(s.Pop() + op);

if (postf[i].GetName() == "-")

s.Push(s.Pop() - op);

if (postf[i].GetName() == "\*")

s.Push(s.Pop() \* op);

if (postf[i].GetName() == "/")

{

if (op == 0.0)

throw Error("Division by zero");

s.Push(s.Pop() / op);

}

if (postf[i].GetName() == "unar-")

s.Push(-op);

}

i++;

}

return(s.Pop());

}

1. *Установка значений переменных*

void Set\_values(vector<Lexeme>& exp)

{

vector<Lexeme> var;

bool key = true;

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

{

if (exp[i].GetType() == VARIABLE)

{

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

{

if (exp[i].GetName() == var[j].GetName())

{

exp[i].SetValue(var[j].GetValue());

key = false;

break;

}

}

if (key)

{

double temp;

cout << "Enter variable value " << exp[i].GetName() << " = ";

cin >> temp;

exp[i].SetValue(temp);

var.push\_back(exp[i]);

}

key = true;

}

}

}