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

Ягилев И.М.

**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**

Необходимо 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. Опционально - наличие математических функций (sin, соs, ln, exp, и т.д.). Программа должна выполнять предварительную проверку корректности выражения и сообщать пользователю вид ошибки

и номера символов строки, в которых были найдены ошибки.

**Метод решения**

1. Разработка шаблонного класса TStack

2. Разбиение исходного арифметического выражения на лексемы (т.е. выделить операнды, операции, переменные)

3. Проверка корректности выражения:

- правильность расстановки скобок.

- пропущены ли операнды или знаки операций

- недопустимые символы

4. Перевод выражения в постфиксную (польскую) запись

5. Вычисление выражения по постфиксной записи

6. Создание консольного интерфейса пользователя.

**Руководство пользователя**

* Пользователь вводит выражение по следующим правилам:
  1. Допускается использование цифр, скобок, четырёх основных операций, и точки;
  2. В качестве переменной может выступать любая строчная английская буква;
  3. Нельзя ставить знак операции после другой операции, исключение – унарный минус;
  4. Нельзя использовать символ пробел;
  5. Скобки должны быть расставлены в соответствии с математическими правилами;
  6. Не допускается запись переменной перед числом без знака умножения. Т.е. (2+y15) – некорректный ввод, (2+y\*15, 2+15\*y, 2+15y) –корректный.
  7. Точки ставятся в соответствии с математическими правилами.
* Программа выдаёт пользователю результат преобразования выражения в удобном для чтения виде;
  1. Если использовались переменные, то попросит ввести значение самой переменной;
* Программа выдаёт пользователю результат преобразования выражения в обратную польскую запись;
* Программа выводит результат;
* Пользователь может продолжить ввод примеров или же выйти из программы, выбрав пункт continue и exit соответственно.

**Описание программной реализации**

**Структуры данных:**

Класс Stack, Lexeme, Solver см. приложение.

vector, string – стандартные структуры языка.

**Алгоритм main\_arithmetic:**

1. Пользователь вводит выражение (строку);
2. Происходим проверка строки на корректность по выше указанным правилам;
3. Перевод строки в вектор лексем - Solver, где каждый элемент – операция, скобки, точка, цифра или переменная;
4. Перевод в обратную польскую запись;
5. Подсчет выражения в постфиксной записи;
6. Вывод результата на консоль;
7. Возможный рекурсивный ввод.

**Алгоритм проверки на корректность:**

1. Проверка корректности ввода скобок включает в себя:

Проверку условия, что первой не может быть закрывающая скобка, а последней – закрывающая.

Объявляется счетчик учета баланса открывающей и закрывающей скобок

Далее, с помощью циклов для открывающей скобки проверяется невозможность постановки операции после скобки (исключение – унарный минус) и невозможность записи значения до скобки.

Для закрывающей скобки проверяется невозможность записи числа после скобки и невозможность поставить операцию до скобки.

Если счётчик равен нулю, то количество открывающих и закрывающих скобок равно.

1. Проверка корректности ввода включает в себя:

Двойной цикл с проверкой на несоответствующие символы. Если находится хотя бы одно несоответствие, то программа кидает исключение.

1. Проверка ввода включает в себя:

Цикл с запретом ввода подобных выражений (2+y15). Условия проверки корректности ввода операций (на первом и последнем местах).

1. Проверка точек представляет собой:

Цикл с возможностью запрета ввода точки после операции.

**Алгоритм перевода строки в вектор Solver:**

Проходим последовательно по строке.

1. Если встречаем операции +,\*,/ или скобки, то вставляем операцию в типе Lexeme в конец вектора Solver.
2. Если встречаем минус:
   * Если минус является первым элементом или стоит после операции или скобок, определяем его унарным и вставляем в конец вектора в виде ‘\_’.
   * В противном случае минус – бинарный, также вставляем его в конец вектора в виде ‘-‘.
3. Если встречаем переменную:
   * Если перед ней находится число, то вставляем знак умножения в конец вектора;
   * Вставляем саму переменную в конец вектора.
4. Если встречаем число или точку:
   * Объявляем счётчик точек для одного числа;
   * Объявляем строку, для подсчета числа;
   * Циклом делаем конкатенацию строки, записываем число с точками;
   * Если точек больше одной, выбрасываем исключение;
   * Далее, создаём число и методом stof присваиваем ему строку;
   * Вставляем число в конец вектора.

**Алгоритм перевода вектора в постфиксную запись:**

1. Объявляем вектор лексем и стек;
2. Если в последовательном цикле попадается число, вставляем его в вектор.
3. В обратном случае создаем флаг учета переменных, если попадается переменная, то просим ввести значение самой переменной, вставляем это значение в вектор;
4. Если обнаружили открывающую скобку, то вставляем её в стек;
5. Если обнаружили закрывающую скобку, то извлекаем символы из стека, пока не обнаружим открывающую скобку, сама скобка извлекается;
6. Если обнаружили унарный минус, вставляем его в стек операций;
7. Если встретили оставшиеся операции, проверяем приоритет

\*/ = 3, +- = 2, ( = 1;

* + Если стек пуст, или символы в нем имеют меньший приоритет текущего, то вставляем символ в стек;
  + В обратном случае извлекаем символы в вектор, пока условие выполняется;

1. Оставшиеся в стеке операции извлекаем в вектор;

**Алгоритм вычисления выражения:**

1. Объявляем стек лексем;
2. Проходим в цикле по вектору в обратной польской записи, если встречаем число, то вставляем его в стек;
3. Если встречаем унарный минус, то меняем знак у последнего значения в стеке;
4. В противном случае:
   * Извлекаем из стека второй и первый операнд;
   * Выполняем действия над операндами исходя из текущей операции;
   * Помещаем в стек результат;
5. Ответ – последняя лексема из стека.

**Подтверждение корректности**

Для подтверждения корректности в программе реализованы два блока тестирования с помощью библиотеки Google Test: тесты стека, тесты арифметики.

**Заключение**

Была успешно реализована программа, корректно вычисляющая арифметические выражения, где в качестве операндов могут выступать переменные и вещественные числа. Допустимые операции: +, -, /, \*, унарный минус. Программа успешно выполняет предварительную проверку корректности выражения и сообщает пользователю вид ошибки, если таковая имеется.

**Приложение**

-------------------------------------------------------------------------------

template <class T>

class Stack {

private:

int stack\_size;

int end;

T\* stack\_ptr;

public:

Stack()

{

stack\_size = 50;

stack\_ptr = new T[stack\_size];

end = -1;

}

Stack(const int tmp\_size)

{

if (tmp\_size == 0)

throw "null size";

stack\_size = tmp\_size;

stack\_ptr = new T[stack\_size];

end = -1;

}

~Stack()

{

delete[] stack\_ptr;

stack\_ptr = 0;

end = -1;

}

// вставка элемента

void push(const T& el)

{

if (full())

{

stack\_size = (stack\_size + 1) \* 2;

T\* tmp\_ptr = new T[stack\_size];

for (size\_t i = 0; i < end + 1; i++)

tmp\_ptr[i] = stack\_ptr[i];

delete[] stack\_ptr;

stack\_ptr = tmp\_ptr;

}

end++;

stack\_ptr[end] = el;

}

// извлечение элемента

T pop()

{

if (empty())

throw "empty";

else

{

stack\_size--;

return stack\_ptr[end--];

}

}

// очистка стека

void clear()

{

end = -1;

delete[] stack\_ptr;

stack\_ptr = new T[stack\_ptr];

}

// просмотр верхнего элемента без удаления

T get\_end()

{

if (empty())

throw "empty";

return stack\_ptr[end];

}

// получение размера стека

T get\_stack\_size()

{

return end + 1;

}

// проверка на пустой стек

bool empty()

{

if (end == -1)

return true;

else

return false;

}

// проверка на полный стек

bool full()

{

if (end >= stack\_size - 1)

return true;

else

return false;

}

};

-------------------------------------------------------------------------------

class Lexeme

{

private:

char operation;

double value;

int type;

public:

Lexeme() = default;

Lexeme(const double& tmp\_value)

{

value = tmp\_value;

type = true;

}

Lexeme(const char& tmp\_operation)

{

operation = tmp\_operation;

type = false;

}

void print\_lexeme()

{

if (type)

cout << value;

else

cout << operation;

}

double get\_value()

{

return value;

}

char get\_operation()

{

return operation;

}

bool get\_type()

{

return type;

}

int priority();

};

-------------------------------------------------------------------------------

class Solver

{

private:

vector <Lexeme> lex;

public:

void string\_to\_lexeme(string& expression);

void lexeme\_to\_reverse();

void print()

{

for (size\_t i = 0; i < lex.size(); i++)

lex[i].print\_lexeme();

}

double calculation();

};

-------------------------------------------------------------------------------

Проверка на корректность:

bool validation(const string& str)

{

if (brackets\_check(str) == false)

throw "brackets error";

else if (symbols\_check(str) == false)

throw "symbols error";

else if (input(str) == false)

throw "input error";

else if (points\_operation\_check(str) == false)

throw "point error";

else

return true;

}

bool brackets\_check(const string& str)

{

if (str[0] == ')' || str[str.length() - 1] == '(')

return false;

int count = 0;

for (size\_t i = 0; i < str.length(); i++)

{

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

{

if (i < str.length())

for (size\_t j = 0; j < oper.length()-1; j++)

if (str[i+1] == oper[j])

throw "operation after braket";

if (i > 0)

for (size\_t j = 0; j < number.length(); j++)

if (str[i-1] == number[j])

throw "number before bracket";

count++;

}

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

{

if (i < str.length())

for (size\_t j = 0; j < number.length(); j++)

if (str[i + 1] == number[j])

throw "number after braket";

if (i > 0)

for (size\_t j = 0; j < oper.length(); j++)

if (str[i - 1] == oper[j])

throw "operation before bracket";

count--;

}

}

if (count == 0)

return true;

}

bool symbols\_check(const string& str)

{

bool flag = false;

for (size\_t i = 0; i < str.length(); i++)

{

string concatenation = oper + number;

for (size\_t j = 0; j < concatenation.length(); j++)

{

if (str[i] != concatenation[j])

{

flag = false;

continue;

}

else

{

flag = true;

break;

}

}

if (flag)

continue;

else

return false;

}

return flag;

}

bool input(const string& str)

{

for (size\_t i = 1; i < str.length(); i++)

{

if (str[i - 1] <= 'z' && str[i - 1] >= 'a' && str[i] >= '0' && str[i] <= '9')

return false;

}

if ((str[0] == '+' || str[0] == '\*' || str[0] == '/') || (str.length() == 1 && (str[0] == '.' || str[0] == '-')))

return false;

else if (str[str.length() - 1] == '+' || str[str.length() - 1] == '-' || str[str.length() - 1] == '\*' || str[str.length() - 1] == '/')

return false;

else

return true;

}

bool points\_operation\_check(const string& str)

{

for (size\_t i = 1; i < str.length(); i++)

{

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

return false;

}

return true;

}

-------------------------------------------------------------------------------

Перевод строки в вектор лексем:

void Solver::string\_to\_lexeme(string& expression)

{

for (size\_t i = 0; i < expression.length(); i++)

{

// operations

for (size\_t j = 0; j < oper.length()-2; j++)

{

if (expression[i] == oper[j])

{

Lexeme l(expression[i]);

lex.push\_back(l);

}

}

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

{

if (i == 0)

{

Lexeme l('\_');

lex.push\_back(l);

}

else

{

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

{

Lexeme l('\_');

lex.push\_back(l);

}

else

{

Lexeme l('-');

lex.push\_back(l);

}

}

}

// variables

for (size\_t j = 10; j <= number.length(); j++)

{

if (expression[i] == number[j])

{

if (i > 0 && expression[i-1] >= '0' && expression[i-1] <= '9')

{

Lexeme m('\*');

lex.push\_back(m);

}

Lexeme l(expression[i]);

lex.push\_back(l);

}

}

// values

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

{

int point\_count = 0;

string str\_number;

int k = 0;

if ((i+k) <= expression.length()) {

while ((expression[i + k] >= '0' && expression[i + k] <= '9') || expression[i + k] == '.')

{

str\_number += expression[i + k];

k++;

if (expression[i + k] == '.')

{

point\_count++;

if (point\_count > 1)

throw "many point error";

}

}

}

i = i + k - 1;

double numb = stof(str\_number);

Lexeme l(numb);

lex.push\_back(l);

}

}

}

-------------------------------------------------------------------------------

Перевод выражения в обратную польскую запись:

void Solver::lexeme\_to\_reverse()

{

vector <Lexeme> reverse;

Stack <Lexeme> op(lex.size());

for (size\_t i = 0; i < lex.size(); i++)

{

if (lex[i].get\_type() == true)

reverse.push\_back(lex[i]);

else

{

bool flag = false;

for (size\_t j = 10; j <= number.length(); j++)

{

if (lex[i].get\_operation() == number[j])

{

flag = true;

cout << "Enter the value of the ";

lex[i].print\_lexeme();

cout << endl;

double variable;

cin >> variable;

reverse.push\_back(variable);

break;

}

}

if (lex[i].get\_operation() == '(')

{

op.push(lex[i]);

continue;

}

else if (lex[i].get\_operation() == ')')

{

while (op.get\_end().get\_operation() != '(')

reverse.push\_back(op.pop());

op.pop();

continue;

}

if (i > 0 && lex[i].get\_operation() == '\_')

op.push(lex[i]);

else

{

while (!op.empty() && op.get\_end().priority() >= lex[i].priority())

reverse.push\_back(op.pop());

if (!flag)

op.push(lex[i]);

}

}

}

while (!op.empty())

reverse.push\_back(op.pop());

lex = reverse;

}

-------------------------------------------------------------------------------

Вычисление выражения:

double Solver::calculation()

{

Stack <Lexeme> stack;

Lexeme lex\_operand1;

Lexeme lex\_operand2;

double operand1, operand2, res;

for (size\_t i = 0; i < lex.size(); i++)

{

if (lex[i].get\_type() == true)

stack.push(lex[i].get\_value());

else if (lex[i].get\_operation() == '\_')

{

res = (-1) \* stack.pop().get\_value();

stack.push(res);

}

else

{

lex\_operand2 = stack.pop();

lex\_operand1 = stack.pop();

operand1 = lex\_operand1.get\_value();

operand2 = lex\_operand2.get\_value();

if (lex[i].get\_operation() == '-')

res = operand1 - operand2;

if (lex[i].get\_operation() == '+')

res = operand1 + operand2;

if (lex[i].get\_operation() == '\*')

res = operand1 \* operand2;

if (lex[i].get\_operation() == '/')

{

if (operand2 == 0)

throw "division error";

res = operand1 / operand2;

}

stack.push(res);

}

}

Lexeme result = stack.pop();

return result.get\_value();

}

-------------------------------------------------------------------------------