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 института ИТММ  
гр. 381703-1

Кудряшов Н. С.

Проверил:

ассистент кафедры МОСТ

Волокитин.В.Д.

Нижний Новгород

2018 г.

Содержание

[Введение 3](#_Toc270962758)

[Постановка задачи 4](#_Toc270962759)

[Руководство пользователя 5-6](#_Toc270962760)

[Руководство программиста 7-10](#_Toc270962761)

[Описание структур данных 7-8](#_Toc270962762)

[Описание алгоритмов 8-10](#_Toc270962763)

[Описание структуры программы 10](#_Toc270962764)

[Заключение 11](#_Toc270962765)

[Литература 12](#_Toc270962766)

[Приложения 13](#_Toc270962767)

# Введение

Представление и вычисление арифметических выражений, проверка различных грамматических конструкций какого-либо языка являются достаточно важными задачами в области компьютерных алгоритмов. Только за вторую половину XX века было разработано множество способов представления арифметического выражения в памяти компьютера и его вычисления. Наиболее известные: алгоритм Рутисхаузера, алгоритм Бауэра и Замельзона, различные рекурсивные способы вычисления выражений, перевод выражения в обратную польскую нотацию и пр.

Такой широкий спектр алгоритмов указывает на действительную важность данной темы, а значит любой программист, как профессионал своего дела, обязан ознакомиться хотя бы с одним из них. В данной лабораторной работе мы рассмотрим алгоритм перевода выражения в обратную польскую нотацию и его последующее вычисление.

# 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. 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.

**Этапы выполнения работы:**Разработка шаблонного класса TStack

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

* правильность расстановки скобок (с использованием стека (TStack) для хранения индексов скобок).
* пропущены ли операнды или знаки операций
* недопустимые символы

1. Перевод выражения в постфиксную (польскую) запись
2. Вычисление выражения по постфиксной записи
3. Создание консольного интерфейса пользователя
4. Написание отчета

Все функции должны тестироваться с помощью Google test-ов.

# Руководство пользователя

# Пользователю программы предоставлен для использования 1 исполняемый файл под названием ''Sample.exe''.

# При открытии исполняемого файла ''Sample.exe'' пользователю будет предложено ввести арифметическое выражение для его последующего расчета. Числа в арифметическом выражении могут быть использованы как дробные, так и целые. Допустимые операции: “+”, ”-”, ”\*”, ”/”, ”(” , ”)”. Допустимы пробелы в арифметическом выражении. В случае синтаксической ошибки в арифметическом выражении программа прекратит вычисление и сообщит точный вид ошибки пользователю. После выполнения расчетов программа предложит повторный ввод или завершение работы программы.

***Пример исполнения***:

1. Hello.  
   Enter the expression to calculate: 4 + 5

Expression result: 9

1. Enter new expression to calculate
2. Exit
3. Hello.  
   Enter the expression to calculate: 4.45 – (5. + 5.98)

Expression result: -6.53

1. Enter new expression to calculate

2. Exit

1. Hello.  
   Enter the expression to calculate: 100 – (-30.3)\*2

Expression result: 260.6

1. Enter new expression to calculate

2. Exit

1. Hello.  
   Enter the expression to calculate: 3 – ( )

“ ( ” can’t go before “ ) “

Please, type the expression in a right way.

1. Enter new expression to calculate

2. Exit

1. Hello.  
   Enter the expression to calculate: 17/\*3

“ / ” can’t go before “ \* “

Please, type the expression in a right way.

1. Enter new expression to calculate

2. Exit

# Руководство программиста

## Описание структуры программы

Программа представлена 4 проектами, объединенными в одно решение

1. ***Arithmetic***

* *Arithmetic.h*Объявление классов *Lexeme* и *Solver*, необходимых для реализации алгоритма, а также их методов.
* *Stack.h*

Объявление шаблонного класса *TStack*, реализующего стек неограниченного размера, реализация методов класса *TStack*.

* *Arithmetic.cpp*

Реализация методов классов *Lexeme* и *Solver.*

1. ***Gtest***

Библиотека Google unit тестов.

1. ***Sample***

Реализация пользовательского интерфейса программы.

1. ***Tests***

* *Test\_arithmetic.cpp*

Unit тесты класса *Solver.*

* *Test\_main.cpp*

Запуск Unit тестов.

* *Test\_stack.cpp*

Unit тесты класса *TStack.*

## Описание структур данных

1. ***Класс TStack***

TStack – класс, реализующий стек неограниченного размера.  
Поля класса:

1. T\* a – указатель на выделенную память для хранения элементов стека.
2. \_size – количество выделенной памяти (число элементов, которое может поместиться в стек до его переполнения)
3. Top – текущее число элементов в стеке. Варьируется от -1 до \_size – 1.

Существует конструктор, как принимающий необходимое число элементов в стеке, так и без него. При отсутствии параметра выделяет для хранения элементов заранее определенное число памяти.

Методы класса:

1. bool is\_full() – проверка стека на полноту
2. void push(T number) – помещение элемента типа T на вершину стека. В случае переполнения стека производит перевыделение большего числа памяти.
3. T pop() – удаление элемента на вершине стека. Выбрасывает исключение при использовании, если стек пуст.
4. T front() – возвращает элемент, находящийся на вершине стека без его удаления.
5. bool is\_empty() – проверка стека на полноту
6. int size() – возвращает текущее число элементов в стеке
7. void clear() – полная очистка стека
8. ***Struct Operation***

Operation – структура, реализующая определение операции, необходимой для алгоритма обратной польской нотации.

Поля структуры:

1. char op – символ операции
2. int prior – приоритет операции
3. ***Класс Lexeme***

Lexeme – класс, реализующий определение лексемы, необходимой для алгоритма обратной польской нотации.

Поля класса:

1. double number – переменная для хранения числа (если заданная лексема - число)
2. Operation op – переменная для хранения операции (если заданная лексема - операция)
3. int check – переменная для определения типа лексемы (число или операция; число – check = 1; операция – check = 0)
4. int amount\_of\_arg – число аргументов операции (в случае, если лексема - это операция; может быть равно 1 в случае унарного минуса и 2 в прочих)

Конструкторы 3 видов: пустой (для возможности выделения памяти через new); принимающий в качестве параметра число double; принимающий в качестве параметра операцию char и число ее аргументов.

Методы класса:

1. bool can\_go\_next(vector <Lexeme>& a,int i) – проверяет, может ли одна лексема идти после другой
2. char get\_oper() – возвращает символ операции лексемы
3. double get\_number() – возвращает числовое значение лексемы
4. int get\_prior() – возвращает приоритет операции лексемы
5. bool is\_op() – проверяет, является ли лексема операцией
6. bool is\_op\_br() – проверяет, является ли лексема открывающейся скобкой
7. bool is\_cl\_br() – проверяет, является ли лексема закрывающейся скобкой
8. bool is\_number() – проверяет, является ли лексема числом
9. bool is\_unary() – проверяет, является ли лексема унарной операцией
10. ***Класс Solver***

Solver – класс, предназначенный для конвертации выражения в обратную польскую запись и его последующего вычисления.

Поля класса:

1. Vector <Lexeme> d – вектор для хранения промежуточных результатов.

Методы класса:

1. void convert\_string\_to\_lexeme(string& s) – преобразует принятое на входе арифметическое выражение в виде строки в массив лексем
2. void convert\_to\_RPN() – преобразует массив лексем в массив лексем, записанных в обратной польской нотации
3. double solve() – вычисление выражения, записанного в обратной польской нотации

## Описание алгоритмов

Описание   алгоритмов, применяющихся в программе   
(реализацию кода см. в Приложении).

***Класс Solver***

1. *void convert\_string\_to\_lexeme(string& s)*  *(1-181)*

Функция предназначена для формирования из строки массива лексем.

* Удаление пробелов из строки *(6 - 12)*
* Семь условных операторов, осуществляющих проверку текущего  элемента строки. Если это не цифра и не символ операции, выбрасывает исключение. *(15 - 179)*
* If  (s[i]>=’0’ && s[i]<=’9’) *(15 – 40)*

Осуществляет сборку цифр в число. Работает с дробными числами. После окончания сборки создает лексему из собранного числа и помещает в вектор d.

* If  (s[i] == ‘/’) *(41 - 47)*

Создает лексему из операции ‘/’ и помещает в вектор d.

* If  (s[i] == ‘\*’) *(48 - 54)*

Создает лексему из операции ‘\*’ и помещает в вектор d.

* If  (s[i] == ‘+’) *(55 - 61)*

Создает лексему из операции ‘+’ и помещает в вектор d.

* If  (s[i] == ‘(’) *(62 - 68)*

Создает лексему из операции ‘(’ и помещает в вектор d.

* If  (s[i] == ‘)’) *(69 - 75)*

Создает лексему из операции ‘)’ и помещает в вектор d.

* If  (s[i] == ‘-’) *(77 - 172)*

Проверяет унарность или бинарность операции ‘-’, проверяет к чему относится ‘-’: к открывающейся скобке или к числу. Собирает в один множественные минусы, если возможно   
(“ -- 2 = 2 ”)

В случае непринадлежности операции ‘-’ к числу   
(..- (38.4+1)..),создает лексему из операции ‘-’ и помещает в вектор d.

1. *void convert\_to\_RPN()*

Функция предназначена для перевода массива лексем в массив лексем, записанный в обратной польской нотации

* Проверка числа скобок *(187 - 205)*
* Проверка синтаксиса при помощи функции can\_go\_next()  
  *(207 - 255)*
* Преобразование выражения в обратную польскую нотацию с использованием стека *(257 - 303)*  
  Рассматриваем поочередно каждый символ:

1. Если этот символ - число, то просто помещаем его в выходную строку.

2. Если символ - знак операции (+, -, \*, /, ) ,( ), то проверяем приоритет данной операции. Операции умножения и деления имеют наивысший приоритет (3). Операции сложения и вычитания имеют меньший приоритет (2). Наименьший приоритет (равен 0) имеет открывающая скобка, закрывающаяся имеет приоритет 1.

Получив один из этих символов, мы должны проверить стек:

а) Если стек все еще пуст, или находящиеся в нем символы   
( находится в нем могут только знаки операций и открывающая скобка) имеют меньший приоритет, чем приоритет текущего символа, то помещаем текущий символ в стек.

б) Если символ, находящийся на вершине стека имеет приоритет, больший или равный приоритету текущего символа, то извлекаем символы из стека в выходную строку до тех пор, пока выполняется это условие; затем переходим к пункту а).

3. Если текущий символ - открывающая скобка, то помещаем ее в стек.

4. Если текущий символ - закрывающая скобка, то извлекаем символы из стека в выходную строку до тех пор, пока не встретим в стеке открывающую скобку, которую следует просто извлечь из стека. Закрывающая скобка также извлекается из стека.

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

1. double solve() *(307 - 378)*

Функция предназначена для вычисления выражения, записанного в форме обратной польской нотации (алгоритм вычисления).

Для реализации этого алгоритма используется стек для чисел.   
В качестве входной строки мы рассматриваем выражение, записанное в обратной польской нотации:

1. Если очередной символ входной строки - число, то кладем его в стек.

2. Если очередной символ - знак операции, то извлекаем из стека два верхних числа (в случае унарного минуса - одно), используем их в качестве операндов для этой операции, затем кладем результат обратно в стек.

Когда вся входная строка будет разобрана в стеке должно остаться одно число, которое и будет результатом данного выражения. Это число функция и возвращает в качестве результата.

# Заключение

В результате выполнения лабораторной работы мы разработали собственный класс TStack, изучили и реализовали алгоритм перевода изначально заданной строки в вид обратной польской нотации, алгоритм вычисления выражений, записанных в форме обратной польской нотации; реализовали обработку ошибок синтаксиса в заданном выражении.

# Литература

# Т. Пратт, М. Зелковиц. Языки программирования: разработка и реализация = Terrence W. Pratt, Marvin V. Zelkowitz. Programming Languages: Design and Implementation. — 4-е издание. — Питер, 2002. — 688 с.

# http://www.interface.ru/home.asp?artId=1492

# Приложения

void Solver::convert\_string\_to\_lexeme(string& s)

{

int i = 0,count = 0;

string temp;

double num;

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

{

while (s[i] == ' ')

{

s.erase(i, 1);

}

}

while (i < s.size())

{

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

{

bool dot = false;

temp = "";

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

{

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

{

string err;

err = "Number has two or more dots in it";

d.clear();

throw err;

}

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

{

dot = true;

}

temp += s[i];

i++;

}

num = stod(temp, 0);

Lexeme a(num);

d.push\_back(a);

continue;

}

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

{

Lexeme a('/');

d.push\_back(a);

i++;

continue;

}

if (s[i] == '\*')

{

Lexeme a('\*');

d.push\_back(a);

i++;

continue;

}

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

{

Lexeme a('+');

d.push\_back(a);

i++;

continue;

}

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

{

Lexeme a('(');

d.push\_back(a);

i++;

continue;

}

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

{

Lexeme a(')');

d.push\_back(a);

i++;

continue;

}

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

{

bool dot = false;

bool unary = false;

temp = "";

count = 0;

count++;

if (i == 0) // -

{

unary = true;

}

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

{

unary = true;

}

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

{

while (i + 1 < s.size() && s[i + 1] == '-')

{

count++;

i++;

}

if (count % 2 == 1 && !unary)

{

Lexeme a('-');

d.push\_back(a);

i++;

continue;

}

else if (count % 2 == 0 && !unary)

{

Lexeme a('+');

d.push\_back(a);

i++;

continue;

}

else if (count % 2 == 1 && unary)

{

temp = "-";

}

// - : '(',number,'-'

if (i + 1 < s.size() && s[i + 1] == '(') // "..-(.."

{

Lexeme a('-',1); //unary minus

d.push\_back(a);

i++;

continue;

}

if (i + 1 < s.size() && s[i + 1] >= '0' && s[i + 1] <= '9') // "..-2.34.."

{

i++;

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

{

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

{

string err;

err = "Number has two or more dots in it";

d.clear();

throw err;

}

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

{

dot = true;

}

temp += s[i];

i++;

}

num = stod(temp, 0);

Lexeme a(num);

d.push\_back(a);

continue;

}

else

{

string err;

err = "Wrong operation order after minus (no number or bracket)";

d.clear();

throw err ; // 2---- or 2----/

}

}

//else

//{

// // throw actually

// Lexeme a('-');

// d.push\_back(a);

// i++;

// continue;

//}

}

else

{

string err;

err = "Incorrect symbol";

d.clear();

throw err;

}

}

}

void Solver::convert\_to\_RPN()

{

int count = 0;

// brackets check

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

{

if (d[i].is\_op\_br())

{

count++;

}

if (d[i].is\_cl\_br())

{

count--;

}

}

if (count != 0)

{

string err;

err = "Wrong amount of brackets";

d.clear();

throw err;

}

//syntax check

if (d[0].is\_op() && !d[0].is\_op\_br() && !d[0].is\_unary())

{

string err;

err = "Expression can not start with operations such as '+','\*','/',')'";

d.clear();

throw err;

}

if (d.back().is\_op())

{

if (!d.back().is\_cl\_br())

{

string err;

err = "Binary operation in the end of the expression";

d.clear();

throw err;

}

}

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

{

if (!d[i].can\_go\_next(d, i))

{

string err = "";

if (d[i].is\_number())

{

err += d[i].get\_number();

err += " can't go before ";

err += d[i + 1].get\_oper();

d.clear();

throw err;

}

else if (d[i].is\_op())

{

err += d[i].get\_oper();

err += " can't go before ";

if (d[i + 1].is\_number())

{

err += d[i + 1].get\_number();

}

else if (d[i+1].is\_op())

{

err += d[i + 1].get\_oper();

}

d.clear();

throw err;

}

}

}

vector <Lexeme> input;

TStack <Lexeme> b;

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

{

if (d[i].is\_number())

{

input.push\_back(d[i]);

} else if (d[i].is\_op())

{

if (d[i].is\_op\_br())

{

b.push(d[i]);

continue;

}

if (d[i].is\_cl\_br())

{

while (!b.front().is\_op\_br())

{

input.push\_back(b.pop());

}

b.pop(); //pop '('

}

gh:

if (b.is\_empty() || b.front().get\_prior() < d[i].get\_prior() || d[i].is\_unary())

{

if (!d[i].is\_cl\_br())

{

b.push(d[i]);

}

}

else

{

while (!b.is\_empty() && b.front().get\_prior() >= d[i].get\_prior())

{

input.push\_back(b.pop());

}

goto gh;

}

}

}

while (!b.is\_empty())

{

input.push\_back(b.pop());

}

d = input;

}

double Solver::solve()

{

TStack <Lexeme> b;

Lexeme a,c;

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

{

if (d[i].is\_number())

{

b.push(d[i]);

}

else

{

if (d[i].is\_unary())

{

a = b.pop();

}

else

{

a = b.pop();

c = b.pop();

}

switch (d[i].get\_oper())

{

case '+':

{

Lexeme temp(c.get\_number() + a.get\_number());

b.push(temp);

break;

}

case '-':

{

if (d[i].is\_unary())

{

Lexeme temp(-a.get\_number());

b.push(temp);

break;

}

else

{

Lexeme temp(c.get\_number() - a.get\_number());

b.push(temp);

break;

}

}

case '\*':

{

Lexeme temp(c.get\_number() \* a.get\_number());

b.push(temp);

break;

}

case '/':

{

if (a.get\_number() == 0.0) // compare with eps. probably??

{

string err;

err = "Division by zero";

d.clear();

throw err;

}

else

{

Lexeme temp(c.get\_number() / a.get\_number());

b.push(temp);

break;

}

}

}

}

}

d.clear();

return b.pop().get\_number();

}