Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное   
учреждение высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

«Разработка транслятора арифметических выражений»

**Выполнил:** студент ИИТММ гр. 0823-1

Романов Александр Анатольевич

**Проверил:** Ассистент каф. МОСТ

Пирова Анна Юрьевна

Нижний Новгород

2016 год

Оглавление

[Введение 3](#_Toc470272273)

[Формулировка задачи 4](#_Toc470272274)

[Руководство пользователя 5](#_Toc470272275)

[Руководство программиста 7](#_Toc470272276)

[Описание структуры программного комплекса 11](#_Toc470272277)

[Описание алгоритмов 12](#_Toc470272278)

[Заключение 15](#_Toc470272279)

[Литература 16](#_Toc470272280)

[Приложение 17](#_Toc470272281)

# Введение

**Интерпрета́тор** — [программа](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) (разновидность [транслятора](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80)), выполняющая *интерпретацию*.

**Интерпрета́ция** — [пооператорный](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от [компиляции](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80), при которой программа транслируется без её выполнения).

Типы интерпретаторов:

**Простой интерпретатор** анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Достоинством такого подхода является мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.

**Интерпретатор компилирующего типа** — это система из [компилятора](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80), переводящего исходный код программы в промежуточное представление, например, в [байт-код](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B9%D1%82-%D0%BA%D0%BE%D0%B4) или [p-код](https://ru.wikipedia.org/wiki/P-%D0%BA%D0%BE%D0%B4), и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая [виртуальная машина](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0)). Достоинством таких систем является большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе). Недостатки — большее требование к ресурсам и требование на корректность исходного кода.

**Обра́тная по́льская нота́ция (ОПН)** — форма записи [математических](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0" \o "Математика)и логических выражений, в которой [операнды](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D0%BD%D0%B4) расположены перед знаками [операций](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Также именуется как *обратная польская запись*, *обратная бесскобочная запись (ОБЗ)*, *постфиксная нотация*, *бесскобочная символика Лукасевича*, *польская инверсная запись*, *ПОЛИЗ*.

# Формулировка задачи

Лабораторная работа заключается в реализации класса для разбора и вычисления арифметических выражений. Выражение может содержать скобки, операции +, -, \*, /, ^ (возведение в степень), || (модуль), константы и символьные переменные (строчные буквы латинского алфавита).

Необходимо реализовать класс, который принимает на вход строку, содержащую выражение, выполняет её разбор, выводит сообщение об ошибке при её обнаружении, выполняет вычисление значения выражения при заданных значениях переменных. Разбор и хранение выражения необходимо осуществлять в обратной польской записи. Необходимо реализовать тесты, содержащие различные типы выражений.

# Руководство пользователя

Для выполнения лабораторной работы необходимо запустить файл Interprier.exe, предварительно записав аргументом объекта input клaсса expression арифметическое выражение(Проект tests -> test\_main.cpp). После компиляции в консоли появится результат, если ввод был совершён корректно, или сообщение об ошибке.

Ниже приведены правила написания арифметического выражения:

Арифметическое выражение записывается в двойных кавычках без пробелов;

Если в выражении присутствуют переменные, после окончания записи выражения необходимо:

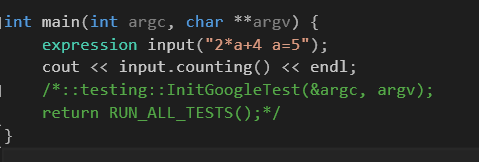
-Поставить пробел

-Написать имя переменной

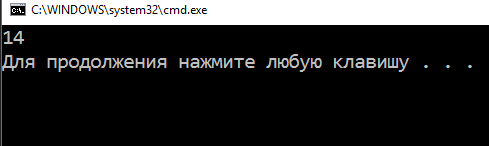
-Поставить знак ‘=’

-Повторять до тех пор, пока таким образом не будут перечислены все переменные

Пример:



После запуска программы на экране появится ответ



# Руководство программиста

**Описание структур данных:**

**Описание структур данных:**

Структуры:

**struct Cnod**; Звено для списка. Лежит в “st\_and\_queue”

**struct \_Node**; Звено для списка. Лежит в “List.h”

**struct variables**; Структура для хранения строк. Лежит в “Expr.h”

Классы:

**class expression;**

**class queue;**

**class stack;**

**class List;**

**class \_Stack;**

Проект **gtest;**

//структура звена шаблонного списка

template <class T>

struct \_Node

{

T data;

\_Node<T> \*next;

\_Node<T>(T i, \_Node \*ptr1) { data = i; next = ptr1; }

//Конструктор для создания нового элемента

};

//Структура для хранения строк

struct variables {

double data;

char \*name\_data;

variables \*next;

};

//Шаблонный класс список

template <class T>

class \_List

{

protected:

int size; //счетчик числа элементов

\_Node<T> \*head;

\_Node<T> \*tail;

public:

\_List() { size = 0; head = tail = NULL; }

~\_List();

T viewData(\_Node<T>\* ptr) { return ptr->data; }

T\* viewPtr(\_Node<T>\* ptr) { return ptr; }

int GetSize() { return size; }

void Print\_list();

bool is\_list\_empty();

int size\_of\_list();

void add\_el\_in\_head(T data);

void add\_el\_in\_tail(T data);

void add\_node(\_Node<T>\* ptr, T data);

T del\_el\_fr\_head();

T del\_el\_fr\_tail();

void del\_node(\_Node<T>\* ptr);

T get\_tail() { return tail->data; }

\_Node<T>\* find\_ptr(T elem);

};

//Шаблонный класс стек

template <class T>

class \_Stack

{

\_List<T> memory;

public:

\_Stack();

T viewData(\_Node<T>\* ptr);

bool is\_stack\_empty();

void push(const T elem);

int size\_of\_stack();

T pop();

T view\_top();

T view\_bot();

T get\_tail();

void del\_el\_fr\_tail();

};

//класс выражение

class expression {

queue ExpStack;

variables \*var;

public:

expression();

expression(char\* input);

void push\_name\_in\_struct(char \*name\_, double data\_);

int prior\_of\_operator(char sign);

bool test\_on\_operator(char input);

bool test\_on\_symbol(char \*input);

double counting();

double find\_el\_var(char \* str);

bool test\_on\_equal\_varnames(char \*str);

bool test\_on\_unar\_minus(char\* input, int pointer, bool abs\_flag);

~expression();

}

};

# Описание структуры программного комплекса

Решение Interprier.sln представлено в виде четырёх проектов:

1. gtest
2. Interprier
3. List
4. tests

Проект test является запускаемым

В проекте gteest лежат декларация и реализация google-тестов.

Проект Interprier создан пустым. Пусть будет.

Проект list содержит декларацию и реализацию классов, позволяющих вычислить арифметическое выражение

Проект tests содержит тесты к классам List; \_Stack; expression;

**“List.h”**; Шаблонный список, шаблонный стек с агрегированным в нём списком.

**“st\_and\_queue.h”**; Стек и очередь для работы со строками

**“Expr.h”**; Класс для вычисления арифметических выражений

Реализация классов для работы со строками не отделена от декларации. Код находится в

st\_and\_queue.h

Трансляция входной строки в обратную польскую запись и вычисление арифметического выражения представлены в файле Expr.cpp

# Описание алгоритмов

Основными алгоритмами программы являются:

**Контруктор с инициализацией:**

expression::expression(char\* input)

Идём вперёд по строке. Читаем очередной символ. Определяем его роль:

* Число(если десятичное, то считываем его до конца, узнаём его длину и записываем в стек для лексем(вместе с ‘.’) ExpStack)
* Модуль либо унарный минус(запоминаем их, поднимаем или опускаем флаги соответственно и записываем в стек для операций OperatorStk. Если найден закрывающий модуль или закрывающая скобка, то из стека операций OperatorStk выталкиваются все операции выражения, стоящего под знаком модуля либо в скобках, и кладутся в ExpStack)
* Операция +, -, \*, /, ^. В соответствии с приоритетом операции, каждая из них записывается в ExpStack в нужном порядке.

Доходим до конца выражения либо до пробела, после которого записаны значения переменных. Если перед пробелом остались символы, бросаем исключение и говорим об ошибке на уровне разбиения на лексемы

Если переменные определены, проверяем на корректность их запись. Также проверяем на корректность запись значений переменных (Наличие одной и только одной ‘.’ В записи десятичного числа).

**Вычисление выражения:**

double expression::counting()

1. Пока стек ExpStack не пуст:
   1. Извлекаем лексему.

* Если лексема – число, записываем его в стек для вычисления выражения st.
* Если лексема – операция, выталкиваем из st два верхних значения, реализуем операцию, и её результат записываем обратно в st.
* Если лексема – корректная переменная, то в стек st записывается значение этой переменной

1. В итоге в st будет находиться одно значение – результат вычислений – если st был заполнен правильно и без лишних лексем. Выталкиваем его и выводим в консоль.
2. Если в стеке осталось больше одного значения, то при вычислении произошла ошибка, и в связи с этим бросается исключение.

**Пример работы алгоритмов:**

Пусть дана строка:

Expression input(“2+3\*a a=4”);

input.counting();

|  |
| --- |
| a |
| 3 |
| 2 |

ExpStack(состояние 1)

|  |
| --- |
| \* |
| + |

OperatorStk

|  |
| --- |
| + |
| \* |
| a |
| 3 |
| 2 |

ExpStack(состояние 2)

St

|  |  |  |
| --- | --- | --- |
| 2 |  |  |
| 2 | 3 |  |
| 2 | 3 | 4 |
| 2 | 12 |  |
| 14 |  |  |

# Заключение

При выполнении лабораторной работы по написанию транслятора для вычисления арифметических операций были разработаны необходимые алгоритмы, позволяющие решить поставленную задачу.

Интерпретатор поддерживает операции ”|”, “^”, “+”, “-“, “\*”, “/”, использование переменных в выражении, использование вещественных чисел.

# Литература

1. Гергель В. П. – Рабочие материалы студента по общему курсу «Методы программированя» (часть 1)
2. <https://ru.wikipedia.org/wiki/> Обратная польская запись
3. <https://ru.wikipedia.org/wiki/> Интерпретатор

# Приложение

Актуально

