МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Отчет по лабораторной работе**

**«Транслятор арифметических выражений»**

**Выполнила:**

студентка группы 3823Б1ФИ1

Пылаева Светлана Алексеевна

**Проверила:**

ассистент кафедры ВВСП ИИТММ

Панова Елена Анатольевна

Рецензент:

Нижний Новгород  
2024

# Содержание

[Содержание 2](#_30j0zll)

[Введение 3](#_1fob9te)

[1](#_3znysh7) Постановка задачи 4

[2](#_2et92p0) Руководство пользователя 6

[3](#_tyjcwt) Руководство программиста 7

[3.1](#_3dy6vkm) Описание структуры программы 7

[3.2](#_1t3h5sf) Описание алгоритмов 8

4[.](#_4d34og8) Эксперименты 11

[Заключение](#_2s8eyo1) 12

[Список литературы](#_17dp8vu) 13

[Приложение](#_3rdcrjn) 14

# Введение

Человеку для решения различных практических задач нередко требуется работать с арифметическими выражениями. Зачастую для обработки арифметических выражений используются трансляторы. Области применения транслятора арифметических выражений могут быть совершенно различны, например, в базах данных и системах аналитики, где требуется выполнение арифметических операций для анализа данных и их представления или в учебных приложениях для изучения математики, программирования и алгоритмов, где необходимо объяснять работу арифметических выражений..

Основа реализации транслятора арифметических выражений состоит из нескольких алгоритмов: анализа выражения (лексического и синтаксического), подразбиение строки на отдельные компоненты(токены или термы), перевода в постфиксную форму, которая необходима для удобного и оптимизированного вычисления и подсчета значения выражения. На самом деле подходов к реализации довольно много и данный список можно изменять и расширять.

Арифметические выражения составляют основную часть большинства программ, так как они используются для представления и манипуляции числовыми данными. Создание транслятора, способного корректно интерпретировать, анализировать и компилировать такие выражения, представляет собой интересную и сложную задачу.

В данной работе представлена реализация транслятора арифметических выражений на языке C++ с пользовательским вводом, поддержкой констант, переменных и чисел с плавающей точкой. Также к проекту прилагаются тесты, проверяющие корректность работы программы.

# Постановка задачи

Цель – изучить базовые алгоритмы необходимые для реализации транслятора арифметических выражений, реализовать транслятор на языке С++, написать тесты для проверки корректности работы программы.

Задачи:

1. Изучить стандартные методы для создания транслятора арифметических выражений:

* лексический анализ строки;
* синтаксический анализ строки;
* перевод арифметического выражения из инфиксной формы в постфиксную форму;
* вычисление значения выражения по его постфиксной форме.

1. Изучить построение конечного автомата, для проведения парсинга строки.
2. Разработать структуру программу, на основе полученных знаний и создать проект с использованием файлов Сmake.
3. Разработать программу с поддержкой:

* вещественных чисел;
* констант pi и e;
* буквенных переменных;
* операций: +, -, \*, /, ^;
* круглых скобок.

1. Реализовать 7 статических библиотек:

* Translator:
* Class Translator (класс, содержащий функции обеспечивающие связную работу всех модулей программы).
* LexicalAnalysis:
* Class LexicalAnalysis (класс содержит одну функцию для анализа строки);
* SyntaxAnalysis:
* Class SyntaxAnalysis (класс, содержит две функции: токенизация стрики и синтаксический анализ строки).
* Postfix:
* Class Postfix (класс, содержащий функцию для перевода набора токенов в постфиксную форму).
* Calculator:
* Class Calculator (класс, содержащий 3 функции: вычисление итогового значения выражения, инициализация переменных, лексический анализ значений переменных, вводимых пользователем).
* Struct:
* Struct Utiliti (структура содержащая необходимые контейнеры для работы с данными поступающими в транслятор);
* Enum termtype (перечисление содержащее типы токенов);
* Class Term (класс, содержащий строку(значение) отдельного токена и его тип).
* Stack:
* Class Stack (класс, содержащий реализацию структуры данных стек (потребуется для перевода в постфиксную форму и вычисления выражения)).

1. Написать тесты, проверяющие корректность работы программы с помощью библиотеки GoogleTest.

# Руководство пользователя

1. Для того чтобы запустить программу необходимо открыть файл Translator.exe.
2. Для вычисления выражения введите его в предлагаемую строку. Ввод может производиться только с использованием символов из алфавита (заглавные и строчные буквы латинского алфавита, цифры от 0 до 9, круглые скобки и арифметические операции (+, -, \*, /, ^)), также выражение можно вводить с любым количеством пробелов.
3. Для подтверждения ввода нажмите “Enter”.
4. Если введенное выражение содержит переменные, то пользователю будет предложено ввести значение каждой переменной.
5. В случае некорректного ввода выражения или значения переменной на экран будет выведено сообщение поясняющее ошибку.
6. В программе поддерживаются следующие константы:
   1. pi = 3.1415;
   2. e = 2.7182.
7. В программе поддерживаются вещественные числа. Примеры возможного ввода:
8. 12.345;
9. .34 (соответствует числу 0.34);
10. 5. (соответствует числу 5.0);
11. Для работы с константами введите выражение включающее в себя название необходимого константного значения именно строчными буквами, иначе она будет воспринята программой как переменная.
12. После вывода ответа можно вводить новое арифметическое выражение.

# Руководство программиста

## Описание структуры программы

Программа разделена на несколько модулей, каждый из которых отвечает за отдельную функциональность. Основная структура представлена следующим образом:

* **Translator.h**:

Содержит класс Translator, который обеспечивает связь между остальными классами. В нем содержатся статический метод execute, который позволяет вычислять произвольное количество выражений за одино выполнение программы, обеспечивает логику программы и запуск необходимых функций с корректными параметрами. Private поля infix и postfix хранят соответственно инфиксную и постфиксную форму введенного арифметического выражения.

* **LexicalAnalysis.h**:

Содержит класс LexicalAnalysis, предоставляющий статический метод LexAnaliysis, выполняющий лексический анализ выражения.

* **SyntaxAnalysis.h**:

Содержит класс Calculator, предоставляющий статические методы:

1. ToTerms, который разбивает введенную строку на токены;
2. Analysis, который проводит синтаксический анализ выражения.

* **Postfix.h**:

Содержит класс Postfix, предоставляющий статический метод ToPostfix, который реализует преобразование выражений из инфиксной формы в постфиксную.

* **Calculator.h**:

Содержит класс Calculator, предоставляющий статические методы:

1. GetVariable для считывания значений переменных, если они есть в выражении;
2. Lex\_Analisis для анализа корректности значения введенной переменной;
3. Calculating для вычисления арифметического выражения в постфиксной форме.

* **Structs.h:**

Содержит структуру Utility, перечисление termtype и класс Term.

1. Utility предоставляет статические поля для хранения набора используемых арифметических операций, приоритетов операций, значений констант и переменных.
2. В termtype объявлены все возможные типы вводимых лексем.
3. Term содержит 2 поля: term типа std::string и type типа typename. Также явно реализованы конструктор инициализатор и конструктор по умолчанию.

* **Stack.h**:

Содержит реализацию структуры данных стек, необходимой для перевода арифметического выражения в постфиксную форму и вычисления значения выражения по постфиксной форме.

## Описание алгоритмов

Ниже приведены основные алгоритмы, используемые в программе:

1. Лексический анализ.
2. Синтаксический анализ.
3. Перевод выражения в постфиксную форму.
4. Вычисление значения по постфиксной форме.

**Лексический анализ.**

Метод LexAnaliysis получает в качестве входного аргумента строку (std::string). Затем производится посимвольная проверка выражения. В строке не могут находиться символы не из алфавита транслятора (заглавные и строчные буквы латинского алфавита, цифры от 0 до 9, круглые скобки и арифметические операции (+, -, \*, /, ^)), а также строка не может быть пустой. В случае несоответствия строки данным условиям кидается исключение, поясняющее на какой позиции располагается первый неверно введенный символ.

**Синтаксический анализ.**

Для корректного проведения синтаксического анализа сначала нужно запустить метод ToTerms, разбивающий введенную строку на токены. Проходя по строке, он создает объекты класса Term, передавая в конструктор выделенную подстроку и тип выражения (число, переменная, константа, операция, скобка), и добавляет их в массив. Также на этапе разбиения выражения на токены проводится частичный синтаксический анализ строки каждого объекта класса Term. Отслеживаются следующие случаи:

1. число не может состоять только из точки;
2. число не может содержать несколько точек;
3. число не может начинаться с 0 (кроме случая когда это вещественное число, например 0.45).

Поэтому объекты класса Term создаются уже с корректным выражением.

На выходе имеем массив термов, где у каждого элемента есть значение в виде строки и его тип.

Затем для проведения второго этапа синтаксического анализа необходимо запустить метод Analysis. В его реализации используется конечный автомат, схема которого приведена на (Рис. 1).

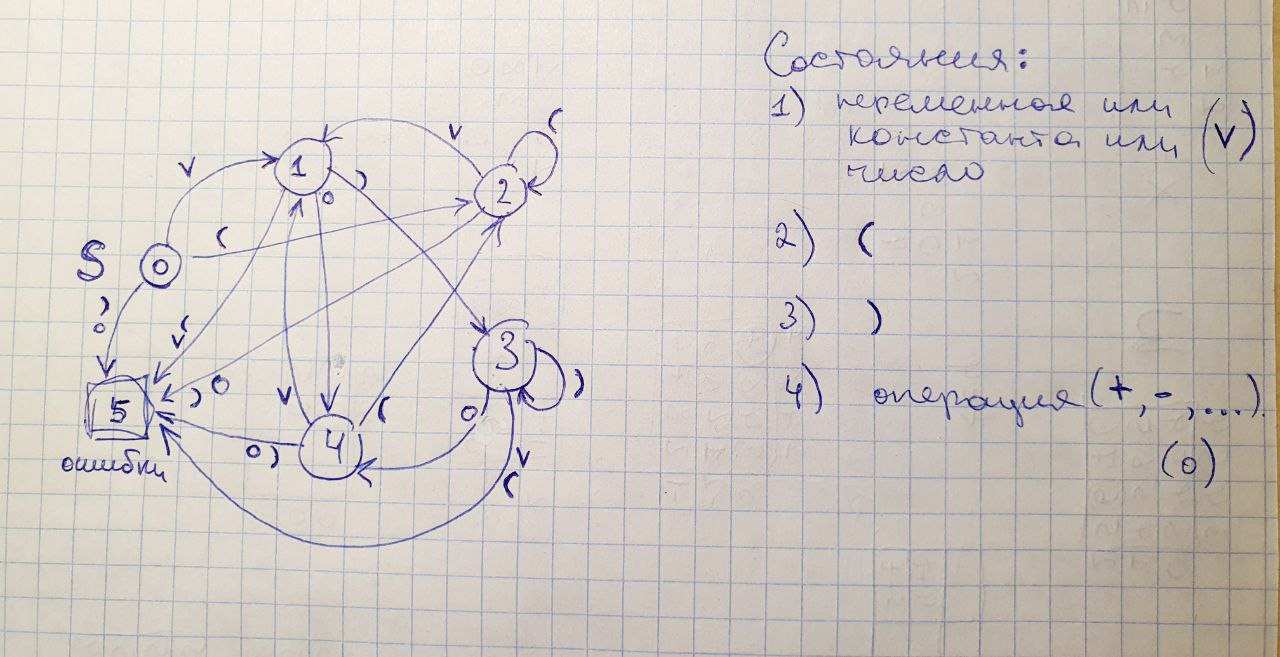


Рис.1 «Конечный автомат»

Также производится подсчет количества открывающих и закрывающих скобок. В случае, если их число не совпадает, то кидается исключение, поясняющее суть ошибки.

**Перевод выражения в постфиксную форму.**

В первую очередь создаются вспомогательные структуры данных: стек для временного хранения операторов и открывающих скобок и вектор объектов класса Term для хранения результата в постфиксной форме. Приоритеты операций хранятся в структуре Utility как статическое поле (std::unordered\_map<std::string, int> priority).

Далее начинает работу сам алгоритм перевода в постфиксную форму и составления массива операндов. Осуществляется проход по вектору токенов и в зависимости от типа текущего токена выполняются следующие действия:

1. Если входящий элемент число, то добавляем его в результирующий вектор постфиксной формы.
2. Если входящий элемент является левой скобкой, то помещаем его в стек.
3. Если входящий элемент операция (+, -, \*, /, ^), то проверяем:

* Если стек пуст или содержит левую скобку в вершине, то добавляем входящую операцию в стек.
* Если входящая операция имеет более высокий приоритет чем вершина, помещаем её в стек.
* Если входящая операция имеет более низкий или равный приоритет, чем вершине, выгружаем элементы стека в результирующий вектор, пока не увидим операцию с меньшим приоритетом или левую скобку на вершине, затем добавляем входящую операцию в стек.

1. Если входящий элемент является правой скобкой, выгружаем стек и добавляем его элементы в очередь, пока не увидим левую круглую скобку. Затем удаляем левую скобку из стека.
2. В конце выражения выгружаем элементы стека в результирующий вектор.

На выходе формируется последовательность токенов, записанных в постфиксной форме, готовая для вычисления ее значения.

**Вычисление значения по постфиксной форме.**

В начале работы алгоритм инициализирует стек для хранения операндов, две переменные (left и right) для проведения промежуточных вычислений т.к. для операций вычитания, деления и возведения в степень важен порядок операндов.

Затем перебираем постфиксное выражение слева направо:

1. Если входящий элемент является оператором, помещаем его в стек;
2. Если входящий элемент является операцией (+, -, \*, /, ^), необходимо получить два последних числа из стека и выполнить соответствующую операцию. Далее помещаем полученный результат в стек;

Таким образом, после обработки всех элементов постфиксного выражения в стеке остается единственный элемент. Он и будет являться результатом вычислений. В конце работы алгоритма возвращаем число, хранящееся в вершине стека.

# Заключение

Таким образом, в ходе работы были изучены и реализованы алгоритмы анализа арифметического выражения, перевода его в постфиксную форму и вычисления значения выражения в постфиксной форме. Разработана программа-транслятор арифметических выражений, реализованная на языке C++. Использование CMake упростило управление зависимостями, сборку и тестирование программы.

Особое внимание было уделено обработке пользовательского ввода: два вида анализаторов (лексический и синтаксический) гарантируют корректность выражения при переходе на этап вычисления итогового значения. Это обеспечивает защиту от ошибок. Например использование недопустимых символов или несоответствие количества скобок.

Проведенные тесты с использованием GoogleTest подтвердили корректность работы всех основных частей программы. В итоге программа продемонстрировала стабильность и соответствие поставленным требованиям.

Разработанный проект обладает гибкостью, что позволит в будущем расширять его функционал. Например, можно будет добавить поддержку новых операций (остаток от деления), функций (вычисление синуса, косинуса и т.д), вычисление факториала и так далее.

Полученные навыки можно будет применять в дальнейших более крупных проектах.

Код проекта лежит в репозитории GIthub. Доступен по ссылке [1].

# Список литературы

1. **Код проекта.**

Доступно по ссылке:

https://github.com/Svetlana-plv/Translator

# Приложение

**Приложение 1.**