**ОТЧЕТ**

по

лабораторной работе

на тему:

**«Вычисление арифметических выражений»**

Выполнил: студент группы 3821Б1ПМ1

**Иванченко А.М.**

**Оглавление:**

[Введение 3](#_Toc1)

[1. Трансляция и вычисление арифметических выражений 4](#_Toc2)

[1.1. Лексический анализ 4](#_Toc3)

[1.2. Преобразование в обратную польскую запись 5](#_Toc4)

[1.3. Вычисление выражения 6](#_Toc5)

[2. Реализация трансляции и вычисления выражений 7](#_Toc6)

[2.1. Класс ArithmeticExpression 7](#_Toc7)

[2.2. Руководство пользователя 7](#_Toc8)

[Заключение 10](#_Toc9)

[Список литературы 11](#_Toc10)

# Введение

За долгую историю развития математики была разработана математическая нотация, которая удобна для понимания и воспроизведения человеку. С тех пор, как электронные вычислительные машины стали повсеместно использоваться для разного рода вычислений, актуальным вопросом стало представление математических (в частности арифметических) выражений на понятном им языке – языке программирования.

Такая задача может возникнуть, например, при создании программы-калькулятора. Известны способы программного решения многих математических задач: это вычисление простейших арифметических выражений, численное интегрирование и дифференцирование и многое другое. Для использования всех этих алгоритмов необходимо реализовать интерфейс программы. Причем желательно, чтобы пользователь мог вводить задачу в привычной ему математической нотации.

Мы не будем рассматривать выражения, возникающие при решения всех подобных математических задач, а остановимся на работе с арифметическими выражениями. В частности, реализуем для языка программирования с++ ввод арифметических выражений, их хранение и вычисление, при этому учитывая собственный опыт в работе с подобными программами и оптимизируя нашу программу под наиболее часто встречающиеся задачи.

## Трансляция и вычисление арифметических выражений

В этой главе описывается метод трансляции вводимых пользователем арифметических выражений в наиболее удобную для вычисления запись.

### Лексический анализ

Стандартной реализацией лексического анализа является использование конечного автомата, определяемого регулярными выражениями. В силу простоты задачи не будем строго определять формальный язык арифметических выражений и ограничимся ссылкой на общепринятую нотацию, на основе которой и построим конечный автомат.

На вход алгоритма подается арифметическое выражение, состоящее из **чисел**, **переменных** (значения которых будут сообщаются при вычислении выражения), **операндов** и **скобок**. Пробелы не несут смысла и не обрабатываются.

**Числа** имеют вид: <знак><целая часть>< . ><дробная часть>

Допускается только знак ‘-’. Знак может быть опущен. Целая часть и дробная часть состоят из цифр. Целая часть и дробная часть могут быть опущены. Точка может быть опущена, если указана целая часть, а дробная часть равна нулю. Случай ввода нескольких точек в одном числе рассматривать не будем, он будет вызывать неопределенное поведение (подробнее в главе 2).

Допустимые **операнды**: +, -, \*, /

**Переменные** состоят из букв латинского алфавита, символа ‘\_’ и цифр. Имя переменной не может начинаться с цифры.

Допускается использование скобок ‘(‘, ‘)’.

Установив такие ограничения, построим конечный автомат для разбиения текста на лексемы. Его задача – для каждого вводимого символа определить тип лексемы, основываясь на информации о типе предыдущего символа. Такая постановка задачи (анализ только одного предшествующего символа) хоть и сужает возможности, но все же обеспечивает определенный выше функционал. Изобразим состояния и переходы между ними в виде графа (рис. 1). Переход между состояний обуславливается очередным поданным символом, соответствующие символы подписаны у ребер графа.

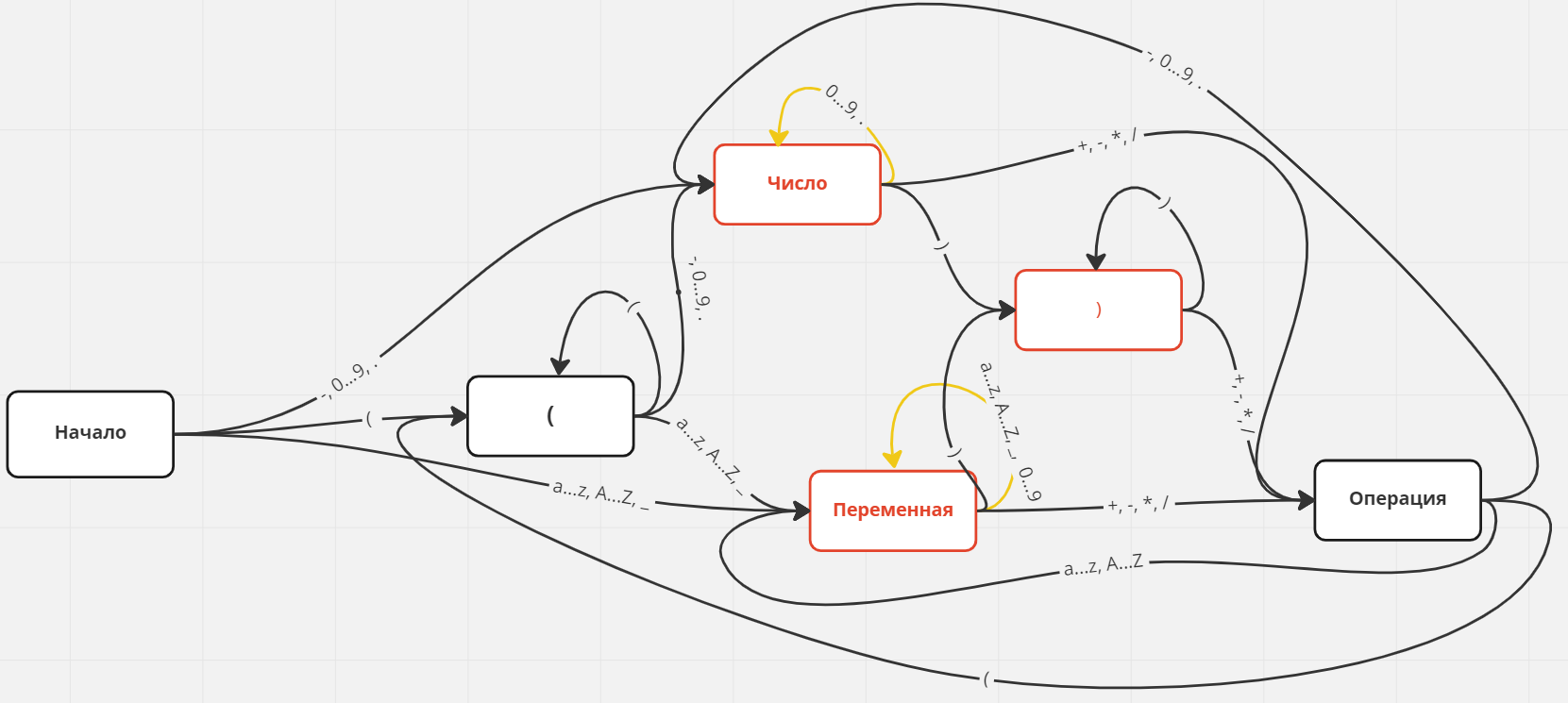


Рисунок 1 – граф состояний и переходов конечного автомата, определяющего тип лексемы очередного символа на основе информации о типе предыдущего символа. Красным помечены конечные состояния. Желтым помечены переходы, при которых происходит накопление символов.

Таким образом, алгоритм действий прост: следуя по графу состояний, копим поступающие символы, если переходим по петле число-число или переменная-переменная (отмечены на графе жёлтым). При нашей постановке задачи только числа и имена переменных могут состоять более чем из одного символа. При переходе по другим ребрам накопленные к моменту перехода добавляем в последовательность лексем и помечаем соответствующим типом.

Если алгоритм закончился в состоянии, не являющимся конечным – введенное выражение некорректно. Также проверим, что открывающих и закрывающих скобок одинаковое количество. Эти две проверки вместе с запретом на недопустимые переходы между состояниями обнаруживают большинство некорректных выражений. Остальные ошибочные выражения будут вызывать исключения на этапе вычисления – это не лучшая практика, но вполне допустимая в нашем случае, когда выражение вводится именно для вычислений и после инициализации обязательно последует вычисление.

Стоит заметить, что время выполнения алгоритма зависит от количества введенных символов линейно. Очевидно, что лучшей асимптотики добиться невозможно, и для лучшей производительности на следующих этапах необходимо также уложиться в линейное время.

### Преобразование в обратную польскую запись

После действий, описанных в предыдущем параграфе, входные данные представляют собой набор лексем, расположенных друг за другом в порядке ввода. Стандартной для математических выражений является инфиксная запись – её мы и ожидаем от пользователя. Но для вычисления выражения более удобна постфиксная запись.

Стандартным решением для вычисления арифметических выражений является обратная польская нотация, которая была разработана австралийским философом и специалистом в области теории вычислительных машин Чарльзом Хэмблином в 1957[1] на основе польской нотации, которая была предложена в 1920 году польским математиком Яном Лукасевичем. Особенностью записи является то, что знак операции всегда расположен после операндов. Этот факт в дальнейшем позволит нам вычислять выражение за линейное время, поэтому требуется осуществить преобразование из инфиксной в обратную польскую запись.

У этой проблемы также есть простое стандартное решение, алгоритм которого можно записать следующим образом (рис. 2)

|  |
| --- |
| Для каждой лексемы ***lex***:  если ***lex*** число:  ***lex*** -> ***результат***  если lex '(':  lex -> стек  если lex ')':  По элементам стека elem пока не '(':  elem -> результат  если lex операция:  По элементам стека elem пока не ')':  если приоритет(lex) <= приоритет(elem):  elem -> результат  стек -> результат |

Рисунок 2 – алгоритм построения обратной польской записи из инфиксой записи

Стоит заметить, что предложенный алгоритм работает за линейное время, то есть достигнутая в прошлом параграфе асимптотика не испорчена.

### Вычисление выражения

Теперь, имея в качестве входных данных набор лексем в обратной польской записи, мы можем с легкостью вычислить выражение за один проход по набору. Алгоритм прост и заключается в следующем: запоминаем операнды, как только встречаем операцию, выполняем её для двух предыдущих операндов. Эти два операнда заменяем на результат операции. Если хранить операнды в стеке, реализация алгоритма тривиальна.

Заметим, что время вычисления зависит линейно от длины выражения, а значит обе операции – создание объекта “арифметическое выражение” и его вычисление выполняются за линейное время. Ещё раз отметим, что лучшей асимптотики в такой задаче добиться невозможно.

## Реализация трансляции и вычисления выражений

Для трансляции и вычисления арифметических выражений была написана программа на языке программирования c++. Для представления арифметических выражений был разработан класс ArithmeticExpression. Он отвечает как за трансляцию, так и за вычисление выражений.

Просто из-за наличия в шаблоне проекта файла stack.h я использовал свою реализацию стека, которую ранее сдавал (см. Stack - Реализация стека за 5 минут) и комментировать не буду, там две с половиной строчки и нечего говорить.

### Класс ArithmeticExpression

Не будем останавливаться на реализации методов класса, так как алгоритмы были описаны в главе 1. Вместо этого кратко опишем интерфейс класса и возможности его использования. Все публичные методы класса представлены на рисунке 3.

|  |
| --- |
| explicit ArithmeticExpression(const string& text);  string getInfix() const;  string getPostfix() const;  double calculate(istream& input = cin, ostream& output = cout); |

Рисунок 3 – публичные методы класса ArithmeticExpression

В конструктор класса передается строка, в которой находится инфиксная запись арифметического выражения. Конструктор может выкинуть исключение, если строка представляет некорректное выражение.

Методы getInfix() и getPostfix() возвращают строку, которая представляет собой инфиксную и обратную польскую запись данного выражения соответственно.

Метод calculate() используется для вычисления значения выражения. При его вызове значение переменных, если таковые присутствуют в выражении, будет введено из потока input (по умолчанию std::cin), сообщения пользователю будут выводиться в поток output (по умолчанию std::cout). Один раз созданное выражение может быть вычислено несколько раз с разными значениями переменных с помощью вызова данного метода.

### Руководство пользователя

Для использования транслятора арифметических выражений необходимо импортировать <postfix.h>.

Создайте объект класса ArithmeticExpression, передав в качестве аргумента строку, содержащую арифметическое выражение в стандартной математической нотации. Строка может содержать:

* Сколько угодно знаков пробела.
* Положительные или отрицательные целые числа или десятичные дроби. Унарный плюс в записи положительных чисел не допускается (можно: “1”, “-1”, нельзя: “+1”). Десятичные дроби записываются с использованием точки; можно опускать целую или дробную часть (но не обе сразу), сохраняя точку (можно: “0.0”, “0.”, “.0”, нельзя: “.” ).
* Имена переменных, состоящие из букв латинского алфавита, символа нижнего подчеркивания “\_” и цифр. Имя переменной не может начинаться с цифры.
* Знаки бинарных операций “ + ”, “ - “, “ \* ”, “ / ”.
* Открывающие и закрывающие скобки “ ( “, “ ) ”.

Создав объект, вы можете получить строку с представлением выражения в инфиксой и постфиксной записях, вызвав методы getInfix() и getPostfix() соответственно. Обратите внимание, что инфиксная запись может отличаться от строки, переданной вами в конструктор количеством знаков пробела.

Чтобы вычислить значение арифметического выражения, вызовите метод calculate(). Следуя появившейся в консоли инструкции, введите значения переменных, которые присутствуют в вашем выражении.

Если на этапе создания объекта или вычисления была выброшена ошибка, ваше выражение некорректно и не соответствует вышеописанным правилам. Убедитесь, что:

* Всем открытым скобкам соответствуют закрытые скобки
* Вы используете допустимые операции явным образом: “2\*x” – можно, “2x” – нельзя.
* Вместо унарного минуса “-a” используется умножение “(-1)\*a”
* Переменные названы допустимым образом (см. выше)
* Для записи десятичной дроби используется точка, а не запятая

Ниже показан пример использования описанного выше функционала (рис.4).

|  |
| --- |
| string str = “2\*x”;    ArithmeticExpression expression(str);  cout << "Инфиксная запись: " << expression.getInfix() << endl;  cout << "Обратная польская запись: "<< expression.getPostfix() << endl;  double res = expression.calculate();  cout << "Результат: "<< res << endl; |

Рисунок 4 – пример использования класса ArithmeticExpression

Так как на практике часто встречаются задачи, в которых необходимо несколько раз вычислять аналогичные выражения с разными константами, рекомендуется вводить их в виде выражения с переменными и вызывать метод calculate() для одного объекта несколько раз. Это позволит уменьшить расходы на трансляцию выражения и может сильно оптимизировать вашу программу.

# Заключение

Для представления и вычисления арифметических выражений на языке программирования c++ был разработан класс, представляющий арифметическое выражение. Он позволяет считывать арифметическое выражение из строки, а затем вычислять его значения. При этом выражение может содержать переменные, значение которых будут подставлены непосредственно при вычислении, что позволяет оптимизировать работу в большом числе случаев.

# Список литературы

1. Hamblin C. L. Language and the Theory of Information : дис. – London School of Economics and Political Science (University of London), 1957.