Министерство образования и науки 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 ПМ1
Смирнов И. К.

Проверил:
преподаватель каф. МОСТ,
Волокитин В.Д.

Нижний Новгород
2023

Содержание

Введение

В данной лабораторной работе мы сосредоточимся на изучении алгоритмов, связанных с переводом выражений в постфиксную форму и последующим вычислением. В ходе выполнения работы мы подробно рассмотрим основные принципы работы с выражениями, стеками и обратной польской записью. Мы уделим особое внимание практическому применению полученных знаний для решения конкретных задач, связанных с обработкой математических выражений в программах. В результате данной лабораторной работы мы приобретем глубокое понимание алгоритмов обработки выражений, а также научимся применять их для разработки эффективных программных решений.

1. Постановка задачи

Цель данной работы — разработка структуры данных Стек и ее использование для расчета арифметических выражений с использованием обратной польской записи (постфиксной формы).

Выполнение работы предполагает решение следующих задач:

- Разработка интерфейса шаблонного класса stack.
- Реализация методов шаблонного класса stack.
- Разработка интерфейса класса ArithmeticExpr для работы с постфиксной формой.
- Реализация методов класса ArithmeticExpr.
- Разработка и реализация тестов для классов stack и ArithmeticExpr на базе Google Test.
- Публикация исходных кодов в личном репозитории на GitHub.

2. Руководство пользователя

Пользователю предлагается использовать программу для вычисления арифметических выражений. Перед вводом непосредственно строки пользователю предоставляются правила ввода выражения, а именно:

1) В выражении не должно быть пробелов.
2) Переменных может быть любое количество, но они имеют только вид x*, где под * подразумевается любое число.
3) В выражении могут быть числа, переменные и представленные функции: sin(x), cos(x), tg(x), ctg(x), ln(x), exp(x)
4) Все операции отделены скобками

После ввода выражения пользователю необходимо ввести значения переменных, если они присутствуют в строке.

Если выражение введено неверно, то пользователю будет выведено сообщение об ошибке и номер символа в строке, где предполагается ошибка, затем пользователь сможет заново ввести выражение. На Рис.1 представлен пример неверного ввода строки.

```
Entry rules:
Enter an expression without spaces
All variables have the form x*, where* is any number;
The expression itself can contain only numbers, variables specified by the rule, and functions (sin(), cos(), tg(), ctg(),
ln(), exp())
All operations are separated by parentheses, that is, an expression of the form 5+-4 is not allowed
Enter a correct expression
6+-5
The resulting infix form is: 6+-5
The resulting postfix form is: 6+5-
The entered string is incorrect
stack is empty
Remark:
If the error is in the '-1' character, it means that the expression ends with an operation or an opening parenthesis
If the error is in the '-2' character, it means that the number of openingand closing brackets is not equal
Enter a correct expression
```

Рис.1. Пример неправильного ввода выражения.

Если выражение введено верно, программа выполнит вычисления и представит пользователю сам результат, а также введенное им выражение в инфиксной форме и полученное выражение в постфиксной форме. (Пример такого исполнения на Рис.2 см. Приложение).

3. Руководство программиста

3.1. Описание структуры программы

Проект программы состоит из файлов: «main_arithmetic.cpp», в котором находится реализация пользовательского приложения; «arithmetic.cpp» и «arithmetic.h», в которых содержатся реализации алгоритмов, работающих с введенным выражением; «stack.h», в котором реализована вспомогательная структура данных — «стек».

Также проект содержит файлы с автоматическими тестами (google tests), проверяющими корректность работы всех функций по отдельности.

3.2. Описание алгоритмов

После ввода пользователем выражения, оно из инфиксной формы записи преобразуется в постфиксную с помощью функции void ToPostfix(). Для этого вызывается функция void Parse(), которая делит полученную строку на лексемы.

Сперва в функции Parse() вызывается вспомогательная функция проверки строки на корректность, написанная для упрощения реализации дальнейшего кода, — функция int StringIsAlmostCorrect(string s). Эта функция возвращает номер символа в строке, в котором обнаружена ошибка, или, если ошибки не найдено, возвращается контрольное значение -128.

Если ошибок не обнаружено, функция начинает разбивать строку на лексемы. Если в строке встретился символ операции или скобки, они добавляются как лексемы в массив, если же символ не операция, то с помощью функции string strNumberOrVar(string infx) (см. Приложение) проверяется, начинается ли на текущем символе число или переменная. Если ни числа, ни переменной не обнаружено, то возвращается пустая строка, иначе возвращается строка, содержащая это число или переменную. Результат функции добавляется в массив лексем.

Далее — в случае неудачи проверки предыдущей функции — с помощью функции string strFunc(string infx) (см. Приложение) проверяется, начинается ли с этого символа реализованная функция. Возвращаемое значение аналогично функции strNumberOrVar(). Если и эта функция вернула пустую строку, то значит что в

исходном выражении обнаружена ошибка и об этом сообщится пользователю. Результат функции так же добавляется в массив лексем.

После разбиения выражения на лексемы выражение преобразовывается в постфиксную форму в функции ToPostfix() по алгоритму:

- Для каждой лексемы в инфиксной форме:

— Если лексема — операнд, поместить ее в постфиксную форму

— Если лексема — открывающая скобка, поместить ее в стек

— Если лексема — закрывающая скобка

• Пока на вершине стека не открывающая скобка

— Извлечь из стека элемент

— Поместить элемент в постфиксную форму

• Извлечь из стека открывающую скобку

— Если лексема — операция

• Пока приоритет лексемы меньше или равен приоритета верхнего элемента стека (приоритет лексем определяется функцией int Priority(string item). В Приложении приведена реализация)

— Извлечь из стека элемент

— Поместить элемент в постфиксную форму

• Поместить лексему в стек

- По исчерпании лексем в инфиксной форме перенести все элементы из стека в постфиксную форму.

В этой же функции, во время работы алгоритма, если лексемой является число, то оно переводится из строки непосредственно в число с помощью функции double Translate(string snum) (см. Приложение). Если лексемой является переменная, то она инициализируется нулем.

После перевода выражения в постфиксную форму пользователю необходимо ввести значения переменных, если они присутствуют в выражении. Далее функция double Calculate(const map<string, double>& value) инициализирует переменные значениями, введенными пользователем, и выполняет вычисление результата по алгоритму:

◆ Для каждой лексемы в постфиксной форме:

— Если лексема — операнд, поместить ее значение в стек

— Если лексема — операция

• Извлечь из стека значения двух операндов

• Выполнить операцию (верхний элемент из стека является правым операндом, следующий за ним — левым)

• Положить результат операции в стек

◆ По исчерпании лексем из постфиксной формы на вершине стека будет результат вычисления выражения.

После вычисления результата по алгоритму он выводится пользователю.

4. Результаты экспериментов

Результатами экспериментов могут служить реализованные и пройденные Google Tests для классов stack и ArithmeticExpr, так как они подтверждают корректность работы программы. Эти тесты позволяют убедиться в правильной реализации алгоритмов работы с арифметическими выражениями в постфиксной форме и эффективности использования стека для обработки данных.

```
[----------] 13 tests from stack
[ RUN      ] stack.can_create_stack
[       OK ] stack.can_create_stack (0 ms)
[ RUN      ] stack.can_do_Is_Empty
[       OK ] stack.can_do_Is_Empty (0 ms)
[ RUN      ] stack.Is_Empty_is_correct
[       OK ] stack.Is_Empty_is_correct (0 ms)
[ RUN      ] stack.can_do_top
[       OK ] stack.can_do_top (0 ms)
[ RUN      ] stack.cant_do_top_when_stack_is_empty
[       OK ] stack.cant_do_top_when_stack_is_empty (0 ms)
[ RUN      ] stack.top_is_correct
[       OK ] stack.top_is_correct (0 ms)
[ RUN      ] stack.can_do_pop
[       OK ] stack.can_do_pop (0 ms)
[ RUN      ] stack.cant_do_pop_when_stack_is_empty
[       OK ] stack.cant_do_pop_when_stack_is_empty (0 ms)
[ RUN      ] stack.pop_is_correct
[       OK ] stack.pop_is_correct (0 ms)
[ RUN      ] stack.get_size_is_correct
[       OK ] stack.get_size_is_correct (0 ms)
[ RUN      ] stack.can_resize
[       OK ] stack.can_resize (0 ms)
[ RUN      ] stack.can_clear
[       OK ] stack.can_clear (0 ms)
[ RUN      ] stack.clear_is_correct
[       OK ] stack.clear_is_correct (0 ms)
[----------] 13 tests from stack (4 ms total)

[----------] 24 tests from Arithmeticexpr
[ RUN      ] Arithmeticexpr.can_create_element
[       OK ] Arithmeticexpr.can_create_element (0 ms)
[ RUN      ] Arithmeticexpr.can_get_infix
[       OK ] Arithmeticexpr.can_get_infix (0 ms)
[ RUN      ] Arithmeticexpr.can_do_PreStringIsCorrect
[       OK ] Arithmeticexpr.can_do_PreStringIsCorrect (0 ms)
[ RUN      ] Arithmeticexpr.PreStringIsCorrect_ret_corect_value_if_string_is_correct
[       OK ] Arithmeticexpr.PreStringIsCorrect_ret_corect_value_if_string_is_correct (0 ms)
[ RUN      ] Arithmeticexpr.PreStringIsCorrect_ret_incorrect_value_if_string_isnt_correct
[       OK ] Arithmeticexpr.PreStringIsCorrect_ret_incorrect_value_if_string_isnt_correct (0 ms)
[ RUN      ] Arithmeticexpr.can_Parse
[       OK ] Arithmeticexpr.can_Parse (0 ms)
```

```
[ RUN      ] Arithmeticexpr.Parse_is_correct
[       OK ] Arithmeticexpr.Parse_is_correct (0 ms)
[ RUN      ] Arithmeticexpr.cant_Parse_when_string_is_not_correct
[       OK ] Arithmeticexpr.cant_Parse_when_string_is_not_correct (1 ms)
[ RUN      ] Arithmeticexpr.can_do_IsNumberOrVar_with_number
[       OK ] Arithmeticexpr.can_do_IsNumberOrVar_with_number (0 ms)
[ RUN      ] Arithmeticexpr.can_do_IsNumberOrVar_with_var
[       OK ] Arithmeticexpr.can_do_IsNumberOrVar_with_var (0 ms)
[ RUN      ] Arithmeticexpr.IsNumberOrVar_with_number_is_correct
[       OK ] Arithmeticexpr.IsNumberOrVar_with_number_is_correct (0 ms)
[ RUN      ] Arithmeticexpr.IsNumberOrVar_with_var_is_corect
[       OK ] Arithmeticexpr.IsNumberOrVar_with_var_is_corect (0 ms)
[ RUN      ] Arithmeticexpr.cant_do_IsNumberOrVar_when_number_isnt_corect
[       OK ] Arithmeticexpr.cant_do_IsNumberOrVar_when_number_isnt_corect (0 ms)
[ RUN      ] Arithmeticexpr.do_IsNumberOrVar_when_there_is_not_number_or_var
[       OK ] Arithmeticexpr.do_IsNumberOrVar_when_there_is_not_number_or_var (0 ms)
[ RUN      ] Arithmeticexpr.can_do_Is_Func
[       OK ] Arithmeticexpr.can_do_Is_Func (0 ms)
[ RUN      ] Arithmeticexpr.Is_Func_is_correct
[       OK ] Arithmeticexpr.Is_Func_is_correct (0 ms)
[ RUN      ] Arithmeticexpr.can_do_ToPostfix
[       OK ] Arithmeticexpr.can_do_ToPostfix (0 ms)
[ RUN      ] Arithmeticexpr.can_get_postfix
[       OK ] Arithmeticexpr.can_get_postfix (0 ms)
[ RUN      ] Arithmeticexpr.postfix_is_correct
[       OK ] Arithmeticexpr.postfix_is_correct (1 ms)
[ RUN      ] Arithmeticexpr.can_calculate
[       OK ] Arithmeticexpr.can_calculate (0 ms)
[ RUN      ] Arithmeticexpr.calculate_is_correct
-32[       OK ] Arithmeticexpr.calculate_is_correct (3 ms)
[ RUN      ] Arithmeticexpr.can_translate_from_string_to_number
[       OK ] Arithmeticexpr.can_translate_from_string_to_number (0 ms)
[ RUN      ] Arithmeticexpr.cant_translate_from_string_to_number_if_string_is_not_correct
[       OK ] Arithmeticexpr.cant_translate_from_string_to_number_if_string_is_not_correct (0 ms)
[ RUN      ] Arithmeticexpr.translate_from_string_to_number_is_correct
[       OK ] Arithmeticexpr.translate_from_string_to_number_is_correct (0 ms)
[----------] 24 tests from Arithmeticexpr (38 ms total)

[----------] Global test environment tear-down
[==========] 37 tests from 2 test cases ran. (46 ms total)
[  PASSED  ] 37 tests.
```

Заключение

Постфиксная форма имеет ряд преимуществ перед инфиксной формой при работе с арифметическими выражениями, что делает ее ценным инструментом для оптимизации и упрощения математических вычислений. Перевод выражений из инфиксной формы в постфиксную и дальнейшее вычисление описывается достаточно простыми алгоритмами; причем работа с постфиксной формой записи проще, чем с инфиксной, что делает использование алгоритмов выгоднее.

Также использование стека для вычисления постфиксных выражений оказалось эффективным и удобным способом обработки данных.

В результате, работа с арифметическими выражениями в постфиксной форме представляет собой эффективный и перспективный подход, который может быть использован для оптимизации выполнения арифметических операций в программах, а также упрощения процесса вычислений.

Приложение

```
Entry rules:
Enter an expression without spaces
All variables have the form x*, where* is any number;
The expression itself can contain only numbers, variables specified by the rule, and functions (sin(), cos(), tg(), ctg(),
ln(), exp())
All operations are separated by parentheses, that is, an expression of the form 5+-4 is not allowed
Enter a correct expression
6*x9-10
Enter a value x9: 5
The resulting infix form is: 6*x9-10
The resulting postfix form is: 6x9*10-
Result of calculation is: 20
```

Рис.2 Пример верной работы программы.

Реализация функции strNumberOrVar():

```
string ArithmeticExpr::strNumberOrVar(string infx) {
	string tmp = "";
	int i = 0, countd=0, counte=0;
	while (infx[i] == 'x'||infx[i] == '0' || infx[i] == '1' || infx[i] == '2' || infx[i] == '3' ||
infx[i] == '4' || infx[i] == '5' || infx[i] == '6' || infx[i] == '7' || infx[i] == '8' || infx[i] == '9' ||
infx[i] == '.'|| (infx[i] == 'e'&&infx[i+1]!='x' && infx[i + 2] != 'p')) {
		tmp += infx[i];
		if (infx[i] == 'e') {
			tmp += infx[++i];
			counte++;
		}
		if (infx[i] == '.') countd++;
		i++;
	}
	if (counte > 1 || countd > 1) throw logic_error("assumed error in the character
number "+to_string(i+1));
	return tmp;
}
```

Реализация функции Priority():

```
int ArithmeticExpr::Priority(string item) {
	int i = 0;
	while (item != priorStr[i]) i++;
	return priorVal[i];
}
```

Реализация функции strFunc():

```
string ArithmeticExpr::strFunc(string infx) {
	string tmp = "";
	int i = 0;
	if (infx[i] == 's' && infx[i + 1] == 'i' && infx[i + 2] == 'n') {
```

```
            tmp = "sin";
        }
        else if (infx[i] == 'c' && infx[i + 1] == 'o' && infx[i + 2] == 's') {
            tmp = "cos";
        }
        else if (infx[i] == 't' && infx[i + 1] == 'g') {
            tmp = "tg";
        }
        else if (infx[i] == 'c' && infx[i + 1] == 't' && infx[i + 2] == 'g') {
            tmp = "ctg";
        }
        else if (infx[i] == 'l' && infx[i + 1] == 'n') {
            tmp = "ln";
        }
        else if (infx[i] == 'e' && infx[i + 1] == 'x' && infx[i + 2] == 'p') {
            tmp = "exp";
        }
        return tmp;
}
```

Реализация функции Translate():

```
double ArithmeticExpr::Translate(string snum) {
        vector<int> fpart;
        vector<int> spart;
        vector<int> power;
        int sign = -1;
        int i = 0;
        while ((snum[i] != '.') && (i < snum.length()) && (snum[i]!='e')) {
            fpart.push_back(int(snum[i]) - 48);
            i++;
        }
        if (snum[i] == '.') {
            if (fpart.size() == 0) fpart.push_back(0);
            i++;
            while (snum[i] != 'e' && snum[i] != '\0') {
                if (snum[i] == '.') throw logic_error("assumed error in the expression
" + snum);
                spart.push_back(int(snum[i]) - 48);
                i++;
            }
            if (spart.size() == 0) spart.push_back(0);
        }
        if (snum[i] == 'e') {
            i++;
            if (snum[i] == '.' || snum[i] == 'e') throw logic_error("assumed error in the
expression " + snum);
```

```
            if (snum[i] == '-') sign = 1;
            else sign = 0;
            i++;
            while (i < snum.length()) {
                  if (snum[i] == '.' || snum[i] == 'e') throw logic_error("assumed error
in the expression " + snum);
                  power.push_back(int(snum[i]) - 48);
                  i++;
            }

      }

      double res = 0;
      int j = fpart.size() - 1;
      while (j >= 0) {
            res += fpart[j] * (pow(10, fpart.size() - j - 1));
            j--;
      }
      j = 0;
      while (j < spart.size()) {
            res += spart[j] * (pow(10, -j - 1));
            j++;
      }
      j = power.size() - 1;
      int e = 0;
      while (j >= 0) {
            e += power[j] * (pow(10, power.size() - j - 1));
            j--;
      }
      if (sign == 1) {
            res /= pow(10, e);
      }
      else if (sign == 0) {
            res *= pow(10, e);
      }
      return res;

}
```