Вычисление выражения

Во-первых, я полностью отказался от прежней версии программы, так как в ней были технические сложности для расширения программы.

Во-вторых, я решил продумать архитектуру, потому что так бы я разобрался в каждом этапе подробно. Проект состоит из функции main, модуля (h-файла и cpp-файла) и двух классов, о которых речь пойдет позже.

Для начала, программа парсит выражение и делит его на терминальные символы: скобки, знаки операций, числа и возвращает вектор в исходном порядке. (Это выполняет функция getMathExpression в файле ParsingString.h)

Т.е. для примера: 4+5+7\*(6+12) функция вернет вектор {4, +, 5, +, 7, \*, (, 6, +, 12, ) }

Далее следует класс ConvertExpressionToPostFix, который превращает распарсенное инфиксное выражение в постфиксное выражение. Там используется функция getPostFixExpression, которая парсит выражение согласно алгоритму с википедии (источники), а также функции для получения приоритета оператора и проверки на то, является ли оператором элемент.

После этого переходим к классу BuildingAST, который содержит в себе структуру узла вершины для построения дерева выражений (как обычно – данные, левый и правый указатели). convertToTree – функция, которая отвечает за построение абстрактного синтаксического дерева (AST) из математического выражения. Она начинает работу с перебора всех лексем выражения, хранящихся в векторе лексем. Если лексема не является оператором, она создает новый узел с лексемой в качестве данных и добавляет его в вектор expressionTree. Если лексема является оператором или функцией, то создается новый узел с оператором в качестве данных и устанавливается в его правого сына на последнюю лексему, добавленную в вектор expressionTree. Затем он устанавливает элемент в его левого сына элемент на предпоследнюю лексему, добавленную к вектору, если таковая имеется, и потом пушит новый узел в вектор. После обработки всех лексем функция получает корневой узел дерева выражений, беря последний элемент вектора и возвращая его в виде общего указателя, чтобы потом выражение можно было бы вычислить. (Как описано на хабре)

В конце все вычисляет функция evaluateExpressionTree. Она начинает обход дерева с корневого узла и рекурсивно переходит к его потомкам. Для каждого узла она вычисляет значение в соответствии с его оператором. На этом все 😊

Описание тестов:

cin1.txt – возведение в отрицательную степень

cin2.txt – Задача Гаусса про сумму от 1 до 100

cin3.txt – умножение на 0

cin4.txt – ввод 1 из примера

cin5.txt – ввод 2 из примера

cin6.txt – случай пример с одним унарным минусом

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

cin7.txt – два унарных минуса подряд

cin8.txt – двойное деление

cin9.txt – пример на максимальное получаемое значение

cin10.txt – пример на отрицательное максимальное получаемое значение

cin11.txt – пример на отрицательное максимальное получаемое значение без скобок

cin12.txt – пример на отрицательное максимальное получаемое значение c минусом у второго множителя

cin13.txt – пример на максимальное получаемое значение с минусом у двух множителей

cin14.txt – пример на обычный приоритет операций 😊

cin15.txt – пример со всеми операциями сразу

Изображение выглядит как текст, Шрифт, рукописный текст, каллиграфия

Автоматически созданное описание

cin16.txt – деление на 0

cin17.txt – 0 в 0 степени

cin18.txt – пример на правоассоциативность степени из вашего замечания

cin19.txt – пример на унарные минусы из вашего замечания