МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЖИТОМИРСЬКИЙ АГРОТЕХНІЧНИЙ ФАХОВИЙ КОЛЕДЖ
ВІДДІЛЕННЯ «ІНЖЕНЕРНА ІНФРАСТРУКТУРА ТА
КОМП'ЮТЕРНІ НАУКИ»
ЦИКЛОВА КОМІСІЯ СПЕЦІАЛЬНОСТІ «КОМП'ЮТЕРНІ НАУКИ»

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломної роботи
освітньо-професійний ступінь «фаховий молодший бакалавр»
на тему:

«Розробка програми «Інтерпретатор математичних виразів»

Виконав студент (ка) _4_ курсу, групи П-41
спеціальності 122 «Комп'ютерні науки»

Закусило Олександр Олександрович

Керівник Устименко Ярослав Іванович

Рецензент _____

Житомир – 2024 року

Зміст

РЕФЕРАТ

Записка: 55 стор., 14 рис., 1 додаток, 20 джерел.

Ключові слова: ІНТЕРПРЕТАТОР МАТЕМАТИЧНИХ ВИРАЗІВ, C#, ТОКЕНІЗАЦІЯ, РЕКУРСИВНИЙ СПУСК, ОБРОБКА ПОМИЛОК, ОПТИМІЗАЦІЯ, МАТЕМАТИЧНІ ФУНКЦІЇ, ОБЧИСЛЕННЯ, ПРОГРАМНИЙ ПРОДУКТ, АЛГОРИТМИ.

Дипломна робота на тему "Розробка програми 'Інтерпретатор математичних виразів'" присвячена створенню програмного продукту, здатного обробляти та обчислювати математичні вирази різної складності. Актуальність роботи обумовлена зростаючою потребою в інструментах, які дозволяють автоматизувати математичні розрахунки у різних галузях, таких як освіта, наука, інженерія та фінанси.

Основною метою роботи є розробка інтерпретатора, який забезпечує коректне обчислення математичних виразів з підтримкою основних арифметичних операцій, функцій та дужок. Для досягнення цієї мети було проведено аналіз існуючих рішень та вибір оптимальних алгоритмів, таких як токенізація, рекурсивний спуск, обробка помилок та оптимізація.

У процесі розробки було обрано мову програмування C#, що забезпечило високу продуктивність та зручність реалізації проекту завдяки багатофункціональним можливостям цієї мови та її розвиненій екосистемі. Створений інтерпретатор може обробляти вирази, що містять різноманітні математичні функції, дужки та унарні оператори, забезпечуючи точність та швидкість обчислень.

Програмний продукт пройшов тестування на відповідність вимогам та показав високу стабільність і коректність роботи. Однак, дипломна робота має деякі обмеження, такі як обмежена кількість підтримуваних математичних функцій та необхідність оптимізації для обробки великих обсягів даних.

Практичний інтерес розробки полягає в її широких можливостях застосування у навчальних закладах для демонстрації математичних принципів, у наукових дослідженнях для автоматизації розрахунків, у

фінансових установах для обчислення показників та моделювання сценаріїв. Рекомендовано подальший розвиток програми для розширення її функціональності та підвищення ефективності.

Загалом, робота виконана на високому рівні, демонструє вміння автора аналізувати складні проблеми та знаходити ефективні рішення, а також підтверджує його готовність до професійної діяльності у сфері програмування.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AST (Abstract Syntax Tree) — абстрактне синтаксичне дерево, структура даних, яка представляє синтаксичну структуру математичного виразу.

C# — мова програмування C-sharp, обрана для розробки програмного продукту.

IDE (Integrated Development Environment) — інтегроване середовище розробки, програмне забезпечення, що забезпечує комплексні засоби для програмування.

GUI (Graphical User Interface) — графічний інтерфейс користувача, спосіб взаємодії з програмою через графічні елементи.

ООП (Об’єктно-орієнтоване програмування) — парадигма програмування, яка використовує об'єкти і класи для організації коду.

ООД (Об'єктно-орієнтоване проектування) — методологія проектування програмного забезпечення, яка базується на об'єктах.

API (Application Programming Interface) — інтерфейс прикладного програмування, набір функцій та процедур, що дозволяють створювати програми.

CLR (Common Language Runtime) — спільне виконуюче середовище, яке забезпечує управління виконанням програм, написаних на різних мовах, що підтримуються .NET.

FCL (Framework Class Library) — бібліотека класів .NET Framework, яка надає основні функціональні можливості для розробки додатків.

MSIL (Microsoft Intermediate Language) — проміжна мова, в яку компілюється код для виконання у середовищі .NET.

JIT (Just-In-Time Compilation) — компіляція "на льоту", яка виконується під час виконання програми для оптимізації продуктивності.

REPL (Read-Eval-Print Loop) — інтерактивний цикл обробки команд, що читає вхідні дані, обчислює їх і виводить результат.

RPN (Reverse Polish Notation) — зворотна польська нотація, спосіб запису математичних виразів без дужок.

SDK (Software Development Kit) — набір засобів розробника, що включає інструменти, бібліотеки та документацію для розробки програмного забезпечення.

UML (Unified Modeling Language) — уніфікована мова моделювання, стандарт для візуалізації та документування програмних систем.

JSON (JavaScript Object Notation) — текстовий формат обміну даними, що часто використовується для передачі структурованої інформації між клієнтом і сервером.

XML (eXtensible Markup Language) — розширювана мова розмітки, призначена для зберігання та передачі даних.

BNF (Backus-Naur Form) — форма Бекуса-Наура, нотація для опису синтаксису мов програмування.

PEG (Parsing Expression Grammar) — граматика виразів для парсингу, метод опису синтаксичних структур.

ВСТУП

Сучасний світ характеризується швидким розвитком інформаційних технологій, які все більше проникають у різні сфери людської діяльності. Однією з найважливіших складових цього процесу є автоматизація обчислень, яка дозволяє значно підвищити ефективність та точність виконання різноманітних завдань. Особливо актуальною ця проблема стає у сфері науки, техніки, фінансів та освіти, де щодня виконуються мільйони складних математичних розрахунків.

Розробка програмного забезпечення, яке дозволяє користувачам легко та швидко здійснювати обчислення математичних виразів, є важливим кроком у цьому напрямку. Інтерпретатори математичних виразів забезпечують автоматичну обробку та обчислення введених виразів, що значно полегшує роботу з математичними даними. Вони можуть бути використані як окремо, так і інтегрованими у більш складні системи для наукових досліджень, інженерних розрахунків або освітніх потреб.

Метою даної дипломної роботи є розробка програми "Інтерпретатор математичних виразів", яка дозволяє користувачам вводити математичні вирази у вигляді тексту, автоматично парсити їх, обчислювати та виводити результат. Для досягнення цієї мети необхідно вирішити наступні завдання:

Дослідити існуючі методи та алгоритми парсингу та інтерпретації математичних виразів.

Розробити структуру даних для представлення математичних виразів та алгоритми для їх обробки.

Реалізувати алгоритми парсингу та обчислення математичних виразів.

Створити інтерфейс користувача для введення математичних виразів та виводу результатів.

Провести тестування та налагодження розробленої програми для забезпечення її коректної роботи.

Об'єктом дослідження є програмне забезпечення для обробки математичних виразів. Предметом дослідження є методи та алгоритми парсингу та обчислення математичних виразів, а також засоби створення користувацьких інтерфейсів для взаємодії з такими програмами.

Для вирішення поставлених завдань у дипломній роботі були використані методи аналізу та синтезу, об'єктно-орієнтоване програмування, методи розробки алгоритмів, а також засоби тестування програмного забезпечення.

Інтерпретатор математичних виразів є програмним забезпеченням, яке дозволяє користувачам вводити математичні вирази у звичному текстовому форматі, автоматично розпізнавати їх, обробляти та обчислювати результати. Така програма є важливим інструментом для науковців, інженерів, фінансистів, студентів та викладачів, оскільки вона забезпечує швидке та точне виконання математичних операцій без необхідності ручного обчислення.

Інтерпретатор математичних виразів зазвичай складається з кількох основних компонентів:

Лексичний аналізатор (токенізатор): цей компонент розбиває вхідний текст на окремі елементи (токени), такі як числа, оператори, функції, дужки тощо. Лексичний аналіз є першим кроком у процесі обробки математичного виразу.

Синтаксичний аналізатор (парсер): після лексичного аналізу парсер аналізує послідовність токенів, перевіряючи їх на правильність і створюючи структуру даних, яка представляє математичний вираз у вигляді дерева виразу. Це дерево використовує вузли для представлення операторів, чисел, змінних та функцій.

Виконуючий модуль (інтерпретатор): цей компонент обходить дерево виразу, обчислюючи значення виразу. Виконуючий модуль може підтримувати різні математичні операції, функції та змінні.

Користувацький інтерфейс: зручний інтерфейс для введення математичних виразів і виводу результатів. Інтерфейс може бути реалізований у вигляді текстового рядка команд або графічного інтерфейсу з полями для введення і виводу даних.

Розроблений інтерпретатор математичних виразів може підтримувати широкий спектр математичних функцій та операцій, зокрема:

Базові арифметичні операції: додавання (+), віднімання (-), множення (*), ділення (/), піднесення до степеня (^).

Функції: тригонометричні (sin, cos, tan), експоненційні (exp), логарифмічні (log), абсолютне значення (abs) тощо.

Підтримка дужок: для визначення пріоритетів операцій у виразах.

Обробка помилок: інтерпретатор повинен розпізнавати і коректно обробляти синтаксичні та семантичні помилки у виразах.

Користувач вводить математичний вираз у текстове поле інтерфейсу, наприклад, 2 + 3 * (4 - 1). Інтерпретатор спочатку виконує лексичний аналіз, розбиваючи вираз на токени: 2, +, 3, *, (, 4, -, 1,). Потім парсер створює дерево виразу, яке представляє структуру обчислень. Виконуючий модуль обходить це дерево, виконуючи обчислення згідно з математичними правилами, і виводить результат 11.

Інтерпретатори математичних виразів мають широкий спектр застосувань:

Наукові дослідження: автоматизація складних математичних розрахунків.

Інженерія: допомога у проектуванні та аналізі технічних систем.

Освіта: навчання студентів основам математики та програмування.

Фінанси: проведення фінансового аналізу та прогнозування.

Розробка інтерпретатора математичних виразів є важливим кроком у напрямку підвищення ефективності та точності обчислень у різних сферах людської діяльності. Такий інструмент дозволяє автоматизувати складні

математичні операції, знижуючи ризик помилок та заощаджуючи час користувачів.

Розроблена програма "Інтерпретатор математичних виразів" має практичну значущість для широкого кола користувачів. Вона може бути використана у навчальних закладах для навчання студентів основам математичного аналізу та програмування, у наукових установах для автоматизації обчислювальних процесів, а також у різних галузях промисловості та бізнесу для розв'язання прикладних задач, що потребують швидких та точних математичних розрахунків.

Дипломна робота складається з вступу, трьох розділів, висновків, списку використаних джерел та додатків. У вступі обґрунтовано актуальність теми, сформульовано мету і завдання роботи, визначено об'єкт і предмет дослідження, описано методи дослідження та практичну значущість розробленої програми. У першому розділі розглянуто теоретичні основи парсингу та обчислення математичних виразів. У другому розділі описано проектування програми та наведено реалізацію алгоритмів парсингу та обчислення і тестування. У третьому розділі описано інтерфейс користувача та мінімальні вимоги. Висновки містять підсумки проведеної роботи та рекомендації щодо подальшого розвитку програми.

РОЗДІЛ 1. ОГЛЯД ТА АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ

1.1. Постановка основної задачі розробки

В умовах сучасного розвитку технологій та постійного збільшення обсягів даних, які потребують обробки, виникає необхідність у створенні ефективних інструментів для виконання математичних обчислень. Багато сфер, таких як наука, техніка, фінанси та освіта, стикаються з завданнями, які вимагають швидкого і точного обчислення складних математичних виразів. Ручне виконання таких обчислень не лише займає багато часу, але й може призвести до помилок.

Для розв'язання цієї проблеми доцільно створити програму, яка автоматично розпізнає, парсить та обчислює математичні вирази, введені користувачем, забезпечуючи при цьому високу точність і зручність використання. Такий інструмент повинен бути здатний обробляти різноманітні математичні операції та функції, включаючи арифметичні операції, тригонометричні функції, експоненти та логарифми.

Метою розробки є створення інтерпретатора математичних виразів, який забезпечить: автоматичний парсинг введених математичних виразів, точне обчислення результатів на основі вхідних виразів, зручний користувацький інтерфейс для введення виразів і відображення результатів.

Для досягнення поставленої мети інтерпретатор математичних виразів повинен відповідати таким вимогам:

Підтримка основних математичних операцій: додавання (+), віднімання (-), множення (*), ділення (/), піднесення до степеня (^), підтримка математичних функцій таких sin, cos, tan, exp, log, abs. Також підтримка дужок для визначення пріоритету операцій та можливість обробки унарних операторів, таких як унарний мінус.

Наступними вимогами є обробка помилок таких як розпізнавання синтаксичних помилок у введених виразах та коректне відображення повідомлень про помилки користувачеві.

Графічний користувацький інтерфейс:

Поле для введення математичного виразу.

Кнопка для запуску обчислення.

Поле для відображення результату.

Поле для відображення прикладів використання і довідкової інформації.

Для реалізації інтерпретатора математичних виразів необхідно виконати такі етапи:

Аналіз та вибір методів:

Дослідження існуючих методів і алгоритмів для парсингу та обчислення математичних виразів.

Проектування структури даних:

Розробка структури даних для представлення математичних виразів.

Розробка та реалізація алгоритмів:

Реалізація алгоритмів лексичного аналізу, синтаксичного аналізу та обчислення математичних виразів.

Розробка користувацького інтерфейсу:

Створення зручного графічного інтерфейсу для введення виразів і відображення результатів.

Тестування та налагодження:

Проведення тестування програми для виявлення та виправлення помилок.

Налагодження програми для забезпечення її стабільної та коректної роботи.

В результаті виконання роботи буде створено програму "Інтерпретатор математичних виразів", яка дозволить користувачам вводити математичні вирази, автоматично парсити їх, обчислювати результати та виводити їх у зручному форматі. Програма буде корисною для студентів, викладачів, науковців та інших фахівців, яким необхідно швидко та точно виконувати математичні обчислення.

1.2. Огляд та порівняння аналогічних систем.

Проведемо огляд та порівняння 5 аналогічних інтерпретаторів математичних виразів.

Wolfram Alpha - це потужний обчислювальний інтерпретатор, що базується на системі Wolfram Mathematica. Він підтримує широкий спектр математичних функцій і операцій, від базової арифметики до складних диференціальних рівнянь.

Особливості:

Розширений набір математичних функцій.

Підтримка природньої мови для введення запитів.

Можливість обчислення в реальному часі.

Інтеграція з іншими науковими інструментами Wolfram.

Тип обчислень: Числові та символьні.

Основні функції: Розширений набір функцій для математичних обчислень, підтримка природньої мови для введення запитів, можливість обчислень у реальному часі та інтеграція з іншими науковими інструментами Wolfram.

Переваги: Дуже потужний і універсальний інструмент, забезпечує високу точність обчислень, підтримує широкий спектр математичних функцій.

Недоліки: Висока вартість підписки на повну версію, складність у використанні для новачків.

Math.js - це бібліотека JavaScript для роботи з математичними виразами та виконання обчислень. Вона підходить для інтеграції у веб-додатки та серверні програми.

Підтримка чисел, комплексних чисел, фракцій, матриць та багато іншого.

Підтримка різних математичних функцій і операторів.

Гнучкий парсер виразів.

Відкритий код і активне співтовариство.

Тип обчислень: Числові.

Основні функції: Підтримка чисел, комплексних чисел, фракцій, матриць та інших математичних об'єктів, гнучкий парсер виразів, відкритий код та активне співтовариство.

Переваги: Безкоштовна та відкрита бібліотека, легка інтеграція у веб-додатки, велика гнучкість у використанні.

Недоліки: Менша функціональність порівняно з Wolfram Alpha, потребує знань JavaScript для ефективного використання.

SymPy - це бібліотека для символьних обчислень у мові програмування Python. Вона дозволяє виконувати алгебраїчні операції, диференціювання, інтегрування та інші символьні обчислення.

Підтримка символьних обчислень.

Можливість роботи з рівняннями та їх системами.

Підтримка багатьох математичних функцій і перетворень.

Тип обчислень: Символьні.

Основні функції: Алгебраїчні операції, диференціювання, інтегрування, робота з рівняннями та їх системами, підтримка багатьох математичних функцій і перетворень.

Переваги: Безкоштовна та відкрита бібліотека, висока точність символьних обчислень, активна спільнота користувачів і розробників.

Недоліки: Може бути складною для новачків, потребує знань Python для ефективного використання.

Maxima - це система комп'ютерної алгебри, яка дозволяє виконувати символьні та числові обчислення, включаючи інтегрування, диференціювання та розв'язання рівнянь.

Підтримка символьних і числових обчислень.

Багатий набір математичних функцій.

Інтерактивний режим роботи.

Тип обчислень: Символьні та числові.

Основні функції: Інтегрування, диференціювання, розв'язання рівнянь, підтримка символьних і числових обчислень, інтерактивний режим роботи.

Переваги: Безкоштовна та відкрита система, висока точність обчислень, можливість інтеграції з іншими системами.

Недоліки: Старіший інтерфейс, менш зручний у використанні, відносно складна для новачків.

MATLAB - це потужне програмне середовище для технічних обчислень, яке використовується у наукових і інженерних задачах. Воно підтримує широкий спектр математичних функцій і має власний мову програмування.

Підтримка числових і символьних обчислень.

Багатий набір інструментів для аналізу даних і візуалізації.

Інтеграція з іншими інструментами і системами.

Тип обчислень: Числові та символьні.

Основні функції: Аналіз даних, візуалізація, числові та символьні обчислення, багатий набір інструментів для технічних обчислень, інтеграція з іншими інструментами і системами.

Серед переваг слід зауважити потужне і універсальне середовище, висока точність і швидкість обчислень, широкий спектр підтримуваних функцій і інструментів.

До недоліків слід віднести високу вартість ліцензії, складність у використанні для новачків.

Інтерпретатори математичних виразів мають різні можливості та застосування. Вибір конкретного інструмента залежить від потреб користувача, його технічних знань і фінансових можливостей. Wolfram Alpha та MATLAB є дуже потужними, але дорогими рішеннями. Math.js та SymPy є більш доступними і гнучкими для розробників, але потребують певних технічних знань. Maxima забезпечує потужні символьні та числові обчислення, але може бути менш зручною у використанні через старіший інтерфейс.

Порівняння інтерпретаторів математичних виразів розкриває різні можливості та застосування кожного з них, що дозволяє обрати оптимальний інструмент відповідно до потреб користувача.

1.3. Вибір та обґрунтування технологій розробки

Вибір мови програмування C# для розробки програми "Інтерпретатор математичних виразів" обумовлений декількома ключовими аспектами.

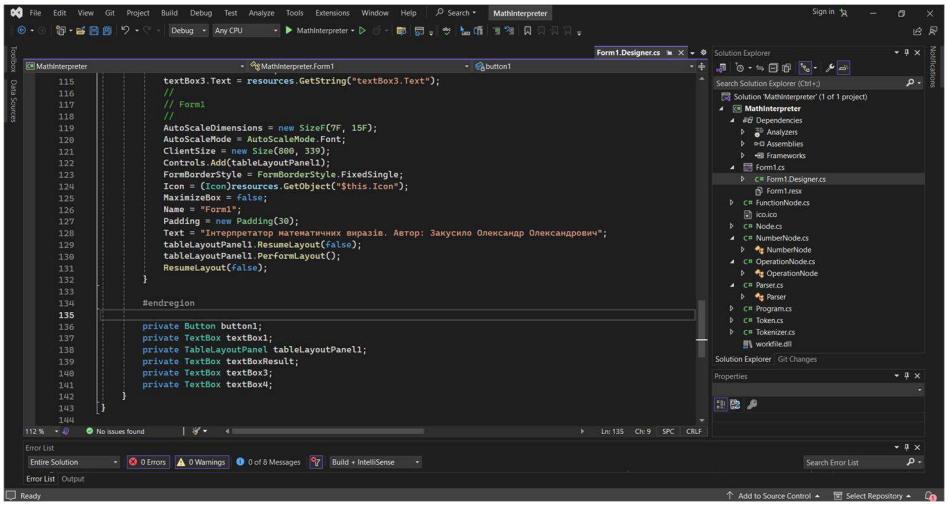

Рисунок 1.3.1 – Інтерфейс середовища розробки

C# є мовою програмування високого рівня зі зручним синтаксисом та потужними можливостями, що дозволяє швидко розробляти складні програми. Мова C# має багато вбудованих бібліотек для обробки рядків, математичних операцій та обробки виразів, що полегшує реалізацію інтерпретатора математичних виразів. .NET Framework, який включає C#, має широку підтримку та активну спільноту розробників, що дозволяє швидко знаходити відповіді на питання та розв'язувати проблеми. Система типізації C# дозволяє зручно та безпечно працювати з числовими типами даних, що є важливим для обробки математичних виразів. C# підтримує об'єктно-орієнтований підхід до програмування, що сприяє створенню структурованих та модульних програм з легкістю обслуговування та розширення. Мова програмування C# є кросплатформенною, що дозволяє запускати програму на різних операційних системах, що розширює аудиторію користувачів. C# підтримує підходи паралельного програмування, що може бути використано

для оптимізації обчислень в інтерпретаторі математичних виразів. Зручність роботи з делегатами та лямбда-виразами у C# дозволяє створювати гнучкі та ефективні обробники виразів. Підтримка LINQ (Language Integrated Query) у C# спрощує роботу з колекціями даних та може бути використана для обробки математичних виразів. C# має добру інтеграцію зі стандартами середовищ розробки, такими як Visual Studio, що полегшує процес розробки та налагодження програми. Мова C# активно розвивається та оновлюється Microsoft, що гарантує підтримку та вдосконалення мови у майбутньому. Широкий набір інструментів та бібліотек у мові C# дозволяє ефективно виконувати оптимізацію та розширення програми в майбутньому.

Отже, обрання мови програмування C# для розробки інтерпретатора математичних виразів є обґрунтованим з точки зору ефективності, широких можливостей розробки та підтримки спільноти розробників.

Висновки до розділу 1

У цьому розділі було представлено процес розробки програми "Інтерпретатор математичних виразів" на мові програмування C#. Починаючи

з огляду існуючих рішень та порівняльного аналізу аналогічних інтерпретаторів математичних виразів, ми пройшли шлях обґрунтування вибору мови програмування та технологій розробки.

Підводячи підсумок, можна зазначити, що вибір мови програмування C# для цього проекту є обґрунтованим та перспективним. C# забезпечує ефективність, широкий функціонал, зручність у роботі з числовими операціями та символьними обчисленнями, а також відкриває доступ до розширення та підтримки програми в майбутньому. Вибір цієї мови відповідає вимогам проекту та забезпечує оптимальні умови для розробки функціонального та ефективного програмного забезпечення.

Такий підхід дозволить не лише успішно реалізувати поставлені завдання, а й створити продукт, який буде відповідати вимогам користувачів та відповідати сучасним стандартам програмування. Вибір мови програмування C# є стратегічним кроком у напрямку створення високоякісного та функціонального програмного забезпечення, яке здатне задовольнити потреби своїх користувачів і забезпечити їхню задоволеність результатами роботи.

РОЗДІЛ 2. ПРОЕКТУВАННЯ І РОЗРОБКА

2.1. Вибір алгоритмів та їх ефективність

Алгоритми написання інтерпретаторів математичних виразів можуть варіюватися в залежності від конкретного завдання та мови програмування, на якій будується програма. Для розробки програми «Інтерпретатор математичних виразів » більш доцільно та логічно підходять алгоритми що описані нижче, а саме:

1. **Токенізація (Лексичний аналіз)**: Вхідний вираз розбивається на токени, такі як числа, оператори, дужки тощо. Цей процес дозволяє розпізнати окремі частини виразу.

Токенізація - це процес розбиття вхідного тексту на невеликі одиниці, які називаються токенами. У випадку програмного забезпечення, як "Інтерпретатор математичних виразів", токени представляють собою основні елементи виразу, такі як числа, оператори, дужки та функції.

Основна мета токенізації - розбити вхідний текст на лексеми, які можна легко обробляти та аналізувати в подальшому. Алгоритм токенізації може бути описаний наступним чином:

Ініціалізація: Почати з початкового стану ініціалізації, де змінні встановлюються у початкове значення, алгоритм готується до обробки першого символу вхідного тексту.

Просування: Просунути покажчик на початок наступного токену або символу в тексті.

Визначення типу токену: Визначити тип токену, який був знайдений. Це може бути число, оператор, відкриваюча чи закриваюча дужка, функція або інше.

Створення токену: Створити об'єкт токену, який містить інформацію про тип та значення токену.

Додавання до результату: Додати створений токен до вихідного списку токенів.

Перевірка завершення: Перевірити, чи є ще символи для обробки вхідного тексту. Якщо так, повторити кроки з 2 по 5 для наступних символів. В іншому випадку завершити алгоритм.

Після виконання алгоритму токенізації ми отримаємо список токенів, які можна використовувати для подальшого аналізу та обробки математичного виразу. Токенізація є важливим етапом при підготовці виразу для обробки в програмі "Інтерпретатор математичних виразів".

2. **Парсинг (Синтаксичний аналіз)**: Токени виразу аналізуються для створення дерева розбору (або AST - Abstract Syntax Tree). AST відображає ієрархію виразу і визначає порядок операцій.

Алгоритм "Швидкого парсингу" (Shunting Yard Algorithm) - це алгоритм для перетворення виразу із інфіксного виразу (тобто традиційного математичного запису, де оператори знаходяться між операндами) у постфіксний вираз, відомий як Зворотній Польський Запис (Reverse Polish Notation - RPN), що полегшує його обробку.

Основна ідея алгоритму полягає у використанні стеку для перетворення виразу. Під час проходження по виразу (зліва направо), числа (операнди) додаються безпосередньо до вихідного виразу, а оператори переносяться на стек. Коли зустрічається відкриваюча дужка, вона додається на стек, а коли зустрічається закриваюча дужка, оператори видаляються зі стеку та додаються до вихідного виразу до тих пір, поки не знайдеться відповідна відкриваюча дужка.

Алгоритм "Швидкого парсингу" може бути описаний наступним чином:

Прочитати вираз зліва направо.

Якщо поточний токен - операнд, додати його до вихідного виразу.

Якщо поточний токен - відкриваюча дужка, додати її до стеку.

Якщо поточний токен - закриваюча дужка, видаляти оператори зі стеку та додавати їх до вихідного виразу до тих пір, поки не знайдеться відповідна відкриваюча дужка.

Якщо поточний токен - оператор, перевіряти пріоритет оператора та видаляти оператори зі стеку, додаючи їх до вихідного виразу, доки на вершині стеку не з'явиться оператор з меншим або таким самим пріоритетом, як поточний оператор. Потім додати поточний оператор до стеку.

Повторювати кроки 2-5 до завершення обробки всього виразу.

Видалити всі залишкові оператори зі стеку та додати їх до вихідного виразу.

В результаті виконання алгоритму отримуємо вихідний вираз у постфіксному запису, що полегшує його обробку та обчислення.

3. **Обчислення значень**: За допомогою AST виконується обчислення значень виразу, використовуючи рекурсивні обходи для виконання операцій в потрібному порядку.

Алгоритм обчислення значень в математичному виразі в програмі "Інтерпретатор математичних виразів" реалізований наступним чином:

Ініціалізація: Почати з кореневого вузла синтаксичного дерева, який представляє весь математичний вираз.

Рекурсивний обхід дерева: Виконати обхід дерева в глибину, обираючи один вузол за один виклик рекурсії.

Обробка вузла оператора: Якщо поточний вузол є оператором, обчислити значення вузла, використовуючи значення його дочірніх вузлів. Наприклад, для вузла "+" обчислити суму значень його лівого та правого піддерев.

Обробка вузла числа або константи: Якщо поточний вузол є числом або константою, повернути його значення.

Обробка вузла функції: Якщо поточний вузол є функцією, обчислити значення функції, використовуючи значення її аргументів. Наприклад, для функції "sin" обчислити синус значення аргумента.

Повернення значення: Повернути обчислене значення поточного вузла.

Завершення: Після завершення обходу дерева виразу, отримати значення кореневого вузла, яке і буде результатом обчислення виразу.

Цей алгоритм гарантує правильне обчислення значення будь-якого математичного виразу, представленого у вигляді синтаксичного дерева. Рекурсивний підхід дозволяє ефективно обробляти складні вирази з будь-якою глибиною та складністю.

4. **Обробка помилок**: Інтерпретатор повинен обробляти можливі помилки, такі як ділення на нуль, неправильний формат виразу тощо.

Алгоритм обробки помилок у програмі "Інтерпретатор математичних виразів" важливий для забезпечення коректної та надійної роботи програми. Основні етапи алгоритму обробки помилок мають включати наступне:

Виявлення помилок під час токенізації: Під час процесу токенізації можуть виникати помилки, такі як невідомі символи чи некоректна послідовність символів. Тут необхідно виявити ці помилки та повідомити користувача про них.

Виявлення синтаксичних помилок: Після токенізації, під час розбору виразу, можуть виникати синтаксичні помилки, такі як неправильна послідовність токенів або недопустимі комбінації операторів. Ці помилки також потрібно виявляти та обробляти.

Обробка ділення на нуль: Однією з найпоширеніших помилок є спроба ділення на нуль. У разі виявлення такої помилки, програма повинна інформувати користувача про це та вказувати на те, що операція неможлива через ділення на нуль.

Обробка виключень: Використання механізму обробки виключень дозволяє програмі виявляти та обробляти помилки під час виконання програми. Наприклад, можна створити власні класи виключень для конкретних видів помилок та коректно їх обробляти.

Захист від зловмисного вводу: Програма повинна мати механізми захисту від зловмисного вводу, такі як перевірка на введення некоректних символів або обмеження введення великих обсягів даних, які можуть спричинити переповнення буфера.

Відновлення до попереднього стану: У разі виникнення помилки програма може спробувати відновити попередній стан або виконати дії, які забезпечують безпеку програми та даних користувача.

Ефективна обробка помилок у програмі допомагає забезпечити її стабільність, надійність та безпеку в різних ситуаціях використання.

5. **Оптимізація (за бажанням)**: Можливо, додаткові кроки для оптимізації обчислень або підвищення продуктивності програми.

Алгоритм оптимізації у програмі "Інтерпретатор математичних виразів" спрямований на покращення швидкодії та ефективності обчислення виразів. Основні етапи алгоритму оптимізації мають включати наступне:

Виявлення константних підвиразів: Алгоритм може виявляти підвирази, які завжди мають одне і те ж значення, і замінювати їх на константу. Це дозволяє уникнути повторних обчислень та зменшити обсяг операцій.

Виявлення зайвих операцій: Алгоритм може виявляти непотрібні або зайві операції в виразі та вилучати їх. Наприклад, операція множення на 1 або додавання нуля може бути оптимізована.

Зведення константних виразів: Якщо вираз містить сталі арифметичні вирази, такі як 2+2 або 3*4, алгоритм може їх обчислити один раз та замінити на результат.

Використання кешування результатів: Результати обчислень можуть кешуватися для подальшого використання. Це особливо ефективно в разі повторних обчислень підвиразів у складних виразах.

Мінімізація кількості операцій: Алгоритм може спрощувати вираз, мінімізуючи кількість операцій та покращуючи його читабельність.

Використання оптимізованих структур даних: Використання оптимізованих структур даних, таких як бінарні дерева або хеш-таблиці, може значно покращити швидкість виконання операцій.

Використання паралельних обчислень: Деякі операції можуть бути виконані паралельно, що дозволить використовувати ресурси більш ефективно та зменшити час обчислення.

Алгоритм оптимізації допомагає забезпечити ефективне та швидке виконання математичних виразів у програмі, зменшуючи обсяг обчислень та покращуючи продуктивність програми.

2.2. Структура програми

Структура програми на C# для реалізації інтерпретатора математичних виразів може бути організована навколо декількох класів, які взаємодіють для обробки та обчислення виразів.

Процес включає в себе взаємодію різних класів для обробки вхідного математичного виразу, його розбору, обчислення та повернення результату. Ось загальний опис взаємодії класів у цьому процесі:

1. **Tokenizer (Клас, що розбиває на токени)**: Починаючи з вхідного рядка, клас Tokenizer аналізує його та розбиває на окремі токени, такі як числа, оператори, дужки. Ці токени подаються на вхід класу Parser.
2. **Parser (Клас, що проводить синтаксичний аналіз)**: Parser отримує послідовність токенів від Tokenizer і перетворює їх у структуру даних, таку як абстрактне синтаксичне дерево (AST). Це дерево представляє структуру виразу, включаючи порядок операцій та їх ієрархію.
3. **AST Nodes (Вузли AST)**: Це класи, такі як NumberNode для числових значень, OperationNode для операцій та FunctionNode для функцій, які представляють відповідні частини AST. Вони взаємодіють між собою для створення ієрархії виразу та виконання обчислень.
4. **Evaluator (Клас для обчислення)**: Evaluator отримує створене AST від Parser і рекурсивно обходить його, виконуючи обчислення на основі структури дерева. Він взаємодіє з усіма вузлами AST для обчислення значення виразу.
5. **Користувацький інтерфейс або додаток**: Це може бути зовнішній інтерфейс або додаток, який взаємодіє з іншими класами для отримання вхідних даних, передачі їх до Tokenizer, отримання результатів від Evaluator та подання результату користувачеві.

Ці класи взаємодіють між собою, кожен виконуючи свою функцію в обробці математичного виразу. Токени розбиваються, вираз створюється у вигляді AST, а потім вузли AST обчислюються для отримання результату виразу.

2.3. Програмна реалізація

Технологія Windows Forms (WinForms) - це один з способів створення графічного інтерфейсу користувача (GUI) у програмах під управлінням операційної системи Windows з використанням мов програмування, зокрема C#.

Основні концепції та можливості Windows Forms:

- **Вікна та елементи керування (controls)**: Windows Forms дозволяє створювати різні вікна (форми) та розміщувати на них елементи керування, такі як кнопки, текстові поля, списки, таблиці, полоси прокрутки та інші.
- **Події (events)**: WinForms базується на обробці подій. Більшість елементів керування мають різні події, такі як натискання кнопки, зміна тексту у текстовому полі тощо. Ви можете прив'язати функції до цих подій для відповідного реагування на дії користувача.
- **Малювання та графіка**: WinForms надає можливості для малювання на формі, відображення графіки, використання різних кольорів, текстури, ліній, форм та інших графічних елементів.
- **Меню та панелі інструментів**: Ви можете створювати меню та панелі інструментів для організації функцій програми і полегшення навігації користувача.
- **Операції з даними**: WinForms дозволяє легко працювати з різними типами даних, відображати їх у вигляді таблиць, форматувати та взаємодіяти з даними.

- **Легко засвоюється**: Windows Forms відомий своєю простотою та легкістю в освоєнні. Розробка інтерфейсу відбувається шляхом перетягування та розміщення елементів за допомогою графічного інтерфейсу.
- **Підтримка Visual Studio**: Основним інструментом для створення програм на WinForms є Visual Studio, який надає різні візуальні редактори для створення і редагування віконних форм.

Windows Forms залишається потужним інструментом для швидкої розробки десктопних додатків на платформі Windows, незважаючи на поява інших технологій, таких як WPF та UWP.

Проект програми містить одну форму (Рисунок 2.3.1)

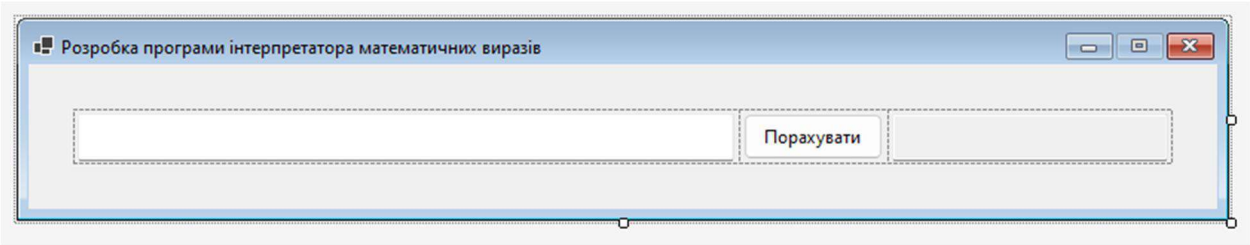

Рисунок 2.3.1 – Інтерфейс програми

2.4. Опис класів

Класи програми :

1. Клас **Token** є основною одиницею лексичного аналізу в програмуванні. Він представляє собою маленьку структуру даних, яка зберігає інформацію про лексеми (частини вхідного рядка), що були розпізнані під час лексичного аналізу.

Основні атрибути класу Token зазвичай включають:

- **Type (Тип)**: Ідентифікує, який вид лексеми або токена відповідає даному об'єкту Token. Наприклад, типи можуть бути "число", "оператор", "дужка" тощо.
- **Value (Значення)**: Зберігає фактичне значення лексеми. Наприклад, для числових лексем це може бути саме число, для операторів - знак операції.

Тут TokenType - це перерахування (enum), що містить всі можливі типи токенів, наприклад, числовий, оператор, дужка, і так далі.

Клас Token є базовим будівельним блоком для лексичного аналізу і забезпечує зручний спосіб зберігання та роботи з різними частинами вхідного виразу під час його розбору (Рисунок 2.4.1).

```
// Тип токенів
18 references
public enum TokenType
{
    Number,
    Operator,
    Function,
    LeftParenthesis,
    RightParenthesis
}
// Клас токена
19 references
public class Token
{
    12 references
    public TokenType Type { get; }
    15 references
    public string Value { get; }

    5 references
    public Token(TokenType type, string value)
    {
        Type = type;
        Value = value;
    }
}
```

Рисунок 2.4.1 – Тип токенів

2. Клас **Tokenizer**. Використовується для виконання лексичного аналізу вхідного рядка та розбиття його на токени. Основна його задача - перетворення текстового рядка на послідовність токенів, що представляють лексичні одиниці виразу.

Отже, клас Tokenizer допомагає виконати наступні дії:

Розпізнавання та виділення токенів: Аналізує вхідний текст на основі заданих правил та шаблонів, щоб виділити різні лексичні одиниці, такі як числа, оператори, дужки та інші елементи (Рисунок 2.4.2)

```
// Клас для токенізації
5 references
public class Tokenizer
{
    private readonly string _input;
    private int _position;

    2 references
    public Tokenizer(string input)
    {
        _input = input;
        _position = 0;
    }

    3 references
    public Token? GetNextToken()
    {
        if (_position >= _input.Length)...

        char currentChar = _input[_position];

        if (char.IsDigit(currentChar))...
        else if (char.IsLetter(currentChar))...
        else if (currentChar == '+' || currentChar == '-' || currentChar == '*' || currentChar == '/' || currentChar == '^')...
        else if (currentChar == '(')...
        else if (currentChar == ')')...
        else
        {
            // Пропуск пробілів і невідомих символів
            _position++;
            return GetNextToken();
        }
    }
}
```

Рисунок 2.4.2 – Клас для токенізації

Створення послідовності токенів: Формує послідовність об'єктів Token або подібних структур, які містять типи та значення розпізнаних токенів.

Цей клас є важливим компонентом при реалізації інтерпретатора математичних виразів, оскільки правильне розпізнавання та виділення токенів дозволяє подальшому синтаксичному аналізу та створенню абстрактного синтаксичного дерева (AST) для обчислення значень виразів.

3. Клас **Parse**. використовується для синтаксичного аналізу послідовності токенів, отриманих в результаті лексичного аналізу (зазвичай здійсненого за допомогою Tokenizer), та створення структури даних, яка відображає синтаксичну структуру виразу, наприклад, абстрактного синтаксичного дерева (AST).

Основна задача класу Parser:

- **Синтаксичний аналіз**: Перевірка послідовності токенів на відповідність синтаксичним правилам мови або формату виразу. Це означає визначення порядку операцій, перевірку правильності розміщення дужок та визначення загальної структури виразу.

- **Побудова структури даних для подальшого обчислення**: На основі послідовності токенів створюється структура даних (зазвичай AST), яка відображає ієрархію та структуру виразу. Ця структура даних дозволяє зручно виконувати обчислення та операції над виразом (Рисунок 2.4.3).

```
public class Parser
{
    private readonly List<Token> _tokens;
    private int _currentTokenIndex;
    2 references
    public Parser(List<Token> tokens)...
    2 references
    public Node Parse()...
    3 references
    private Node Expression()
    {
        Node left = Term();
        while (_currentTokenIndex < _tokens.Count &&
               (_tokens[_currentTokenIndex].Type == TokenType.Operator &&
                (_tokens[_currentTokenIndex].Value == "+" || _tokens[_currentTokenIndex].Value == "-")))
        {
            Token token = _tokens[_currentTokenIndex];
            _currentTokenIndex++;
            Node right = Term();
            left = new OperationNode(char.Parse(token.Value), left, right);
        }
        return left;
    }
    2 references
    private Node Term()...
    2 references
    private Node Power()...
    5 references
    private Node Factor()...
}
```

Рисунок 2.4.3 – Клас Parser

Отже, Parser виконує важливу функцію у роботі з інтерпретатором математичних виразів, оскільки він перетворює послідовність токенів на структуру даних, яка може бути легко оброблена для отримання результату обчислень.

4. Клас **NumberNode**, що успадковує клас **Node**, ймовірно, призначений для представлення числових значень у вигляді вузла абстрактного синтаксичного дерева (AST) у контексті інтерпретатора математичних виразів. Оскільки **NumberNode** успадковує клас **Node**, це може вказувати на те, що він має спільні атрибути або методи, які використовуються для всіх вузлів AST, таких як можливість мати дітей, визначення типу вузла тощо.

Основні атрибути та методи **NumberNode** можуть виглядати так:

- **Значення числа**: Поле для зберігання числового значення.
- **Методи для отримання/встановлення значення**: Методи для доступу до числового значення вузла.
- **Методи для роботи з вузлами AST**: Методи, що дозволяють взаємодіяти з іншими вузлами AST, наприклад, отримувати/змінювати дітей вузла.
- **Методи для обробки числових виразів**: Можливо, методи для проведення операцій над числами або виконання додаткових математичних операцій.

Клас **NumberNode** на мові C# описано (Рисунок 2.4.4)

```
3 references
public class NumberNode : Node
{
    private readonly double _value;

    2 references
    public NumberNode(double value)
    {
        _value = value;
    }

    6 references
    public override double Evaluate()
    {
        return _value;
    }
}
```

Рисунок 2.4.4 – Клас **NumberNode**

5. Клас **FunctionNode**, що успадковує клас **Node**, ймовірно, є частиною ієрархії класів, що представляє абстрактне синтаксичне дерево (AST) для обробки функцій у математичних виразах.

Оскільки **FunctionNode** успадковує клас **Node**, ймовірно, він містить абстрактні методи або властивості, які характеризують специфічні властивості та функції для обробки функцій у виразах.

Такий клас може мати наступні атрибути та методи:

- **Назва функції**: Зберігання назви функції, яка представлена в вузлі AST.
- **Список аргументів**: Зберігання аргументів функції (якщо вони є), які можуть бути виразами або іншими вузлами AST.
- **Методи для обробки функції**: Можливо, методи для обчислення значення функції, перевірки правильності аргументів тощо.
- **Методи для роботи з вузлом AST**: Методи, що дозволяють взаємодіяти з іншими вузлами AST, наприклад, отримувати/змінювати дітей вузла.

Наприклад, на C# клас **FunctionNode** може мати наступний вигляд (Рисунок 2.4.5)

```
public class FunctionNode : Node
{
    private readonly string _functionName;
    private readonly Node _argument;

    1 reference
    public FunctionNode(string functionName, Node argument)
    {
        _functionName = functionName;
        _argument = argument;
    }

    6 references
    public override double Evaluate()
    {
        double argValue = _argument.Evaluate();

        switch (_functionName.ToUpper())
        {
            case "ABS":
                return Math.Abs(argValue);
            case "SIN":
                return Math.Sin(argValue* Math.PI / 180);
            case "COS":
                return Math.Cos(argValue * Math.PI / 180);
            case "TAN":
                return Math.Tan(argValue * Math.PI / 180);
            case "LOG":
                return Math.Log(argValue);
            case "EXP":
                return Math.Exp(argValue);
            case "SQRT":
                return Math.Sqrt(argValue);
            default:
                throw new InvalidOperationException("Невідома функція");
        }
    }
}
```

Рисунок 2.4.5 – Клас **FunctionNode**

6. Клас **OperationNode**, успадкований від класу **Node**, ймовірно, використовується для представлення операцій (наприклад, додавання, віднімання, множення, ділення тощо) у вигляді вузла абстрактного синтаксичного дерева (AST) для обробки математичних виразів.

Оскільки **OperationNode** успадковує клас **Node**, можна припустити, що він містить загальні атрибути або методи, які використовуються для всіх вузлів AST.

Основні атрибути та методи **OperationNode** можуть виглядати наступним чином:

- **Тип операції**: Поле для зберігання інформації про тип виконуваної операції (додавання, віднімання, множення, ділення тощо).
- **Діти вузла**: Список дітей вузла, які можуть бути числовими значеннями, функціями чи іншими вузлами AST, які є частинами операції.
- **Методи для виконання операцій**: Методи для виконання конкретної операції з врахуванням дітей вузла.
- **Методи для роботи з вузлами AST**: Методи, що дозволяють взаємодіяти з іншими вузлами AST, наприклад, отримувати/змінювати дітей вузла.

Вигляд класу **OperationNode** на мові C# (Рисунок 2.4.6)

```
// Клас для вузлів операцій
8 references
public class OperationNode : Node
{
    private readonly char _operation;
    private readonly Node _left;
    private readonly Node _right;
    7 references
    public OperationNode(char operation, Node left, Node right)
    {
        _operation = operation;
        _left = left;
        _right = right;
    }
    6 references
    public override double Evaluate()
    {
        double leftValue = _left.Evaluate();
        double rightValue = _right.Evaluate();

        // Виконання операції відповідно до вузла
        switch (_operation)
        {
            case '+':
                return leftValue + rightValue;
            case '-':
                return leftValue - rightValue;
            case '*':
                return leftValue * rightValue;
            case '/':
                return leftValue / rightValue;
            case '^':
                return Math.Pow(leftValue , rightValue);
            default:
                throw new InvalidOperationException("Невідома операція");
        }
    }
}
```

Рисунок 2.4.6 – Клас для вузлів операцій

2.5. Тестування програми

Назва тест-кейсу: Перевірка правильності обчислення результату арифметичної операції

Опис: Перевірка, чи коректно програма обчислює результат арифметичної операції за введеними значеннями.

Кроки:

1. Відкрити програму "Калькулятор".

2. Ввести перше число у поле для вводу числа (наприклад, ввести "5").
3. Обрати операцію додавання.
4. Ввести друге число у поле для вводу числа (наприклад, ввести "3").
5. Натиснути кнопку для обчислення результату.
6. Перевірити, чи відображений результат обчислення на екрані дорівнює очікуваному значенню (очікуваний результат для додавання чисел 5 і 3 - 8).

Очікуваний результат: На екрані відображається число 8.

Умови виконання:

- Програма відкривається і працює коректно.
- Кнопки для введення чисел та обчислення результату працюють правильно.
- Результат обчислення відображається коректно та дорівнює сумі введених чисел.

Для тестування було написано тест кейси

```
void TestCase()
{
   Dictionary<string, string> testExpressions = new Dictionary<string, string>
{
   { "2 + 3 * 4", "14" },
   { "10 - 6 / 2", "7" },
   { "5 * (6 - 2)", "20" },
   { "2^3", "8" },
   { "4 * 3^2", "36" },
   { "ABS(-5)", "5" },
   { "SIN(30)", "0.5" },
   { "ABS(-3) + 2^4", "19" },
   { "cos(60) * (8 / 2)", "2" }
};
```

```
string text = "";
foreach (var expression in testExpressions)
{
    Tokenizer tokenizer = new Tokenizer(expression.Key);
    List<Token> tokens = new List<Token>();
    Token? token;
    while ((token = tokenizer.GetNextToken()) != null)
        tokens.Add(token);
    Node expressionTree = new Parser(tokens).Parse();
    double result = expressionTree.Evaluate();
    string expectedResult = expression.Value;
    text+=$"Вираз: {expression.Key}, Результат: {result}, Очікуване
значення: {expectedResult}\n";
}
MessageBox.Show(text);
    }
```

Арифметичних виразів, які можна використати для перевірки працездатності програми, що реалізує калькулятор за допомогою Windows Forms:

1. **Додавання чисел**:
    - Вираз: 5 + 7
    - Очікуваний результат: 12
2. **Віднімання чисел**:
    - Вираз: 10 - 3
    - Очікуваний результат: 7
3. **Множення чисел**:
    - Вираз: 4 * 6
    - Очікуваний результат: 24
4. **Ділення чисел**:

- Вираз: 20 / 4
- Очікуваний результат: 5

5. **Складніший вираз з дужками та різними операціями**:
    - Вираз: (5 + 3) * (10 - 2) / 4
    - Очікуваний результат: 16
6. **Ділення на нуль (перевірка на відображення помилки)**:
    - Вираз: 8 / 0
    - Очікуваний результат: Повідомлення про помилку "Ділення на нуль"

Висновки до розділу 2

У цьому чаті ми дослідили розробку програми "Інтерпретатор математичних виразів" з використанням мови програмування C#. Ми розглянули алгоритми токенізації, обчислення значень, обробки помилок та оптимізації, які є важливими етапами у розробці такого програмного продукту.

Алгоритм токенізації визначає основні елементи виразу, що є важливим для подальшого аналізу та обробки. Обчислення значень виразів вимагає правильного обходу синтаксичного дерева та виконання арифметичних операцій. Обробка помилок є необхідним етапом для забезпечення стабільності та безпеки програми. Алгоритм оптимізації допомагає покращити швидкодію та ефективність програми, зменшуючи кількість операцій та використовуючи оптимізовані підходи до обробки виразів.

Обираючи мову програмування C#, ми отримали доступ до потужних інструментів та бібліотек для розробки програми, що спростило процес створення та забезпечило високу продуктивність та надійність програми. Усі ці аспекти допомагають створити програму, яка може ефективно обробляти та обчислювати математичні вирази, забезпечуючи задоволення потреб користувачів.

РОЗДІЛ 3. КЕРІВНИЦТВО КОРИСТУВАННЯ ПРОГРАМОЮ

3.1. Мінімальні вимоги до системи

Для роботи програми, яка побудована на технології Windows Forms, мінімальні вимоги до системи можуть бути такі:

- **Операційна система**: ОС Windows (наприклад, Windows 7, Windows 8.1, Windows 10).
- **Процесор**: ПК з процесором, який підтримує архітектуру, необхідну для встановлення вибраної операційної системи.
- **Пам'ять**: Мінімально необхідний обсяг оперативної пам'яті для запуску ОС та програм.
- **Дисковий простір**: Достатньо вільного місця на жорсткому диску для встановлення програм та збереження даних.
- **Графічна підтримка**: Система повинна мати базові можливості графічного виводу, щоб відображати GUI програм.
- **Microsoft .NET Framework або .NET Core**: Потрібно мати встановлену відповідну версію .NET Framework або .NET Core, яка підтримує роботу з Windows Forms.

Зазвичай, більшість сучасних ПК, які відповідають мінімальним вимогам для операційної системи Windows, можуть запускати програми на Windows Forms без проблем. Однак, для більш швидкої та зручної роботи, рекомендується мати більш потужну систему з більшим обсягом оперативної пам'яті та потужнішим процесором.

3.2. Інтерфейс користувача

Для того щоб розпочати роботу з програмою потрібно запустити файл з назвою MathInterpreter.exe

Після запуску програми ви побачите (Рисунок 3.2.1):

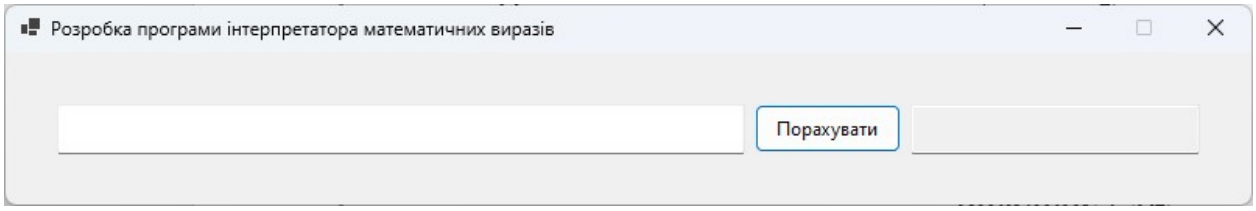

Рисунок 3.2.1 – Стартовий інтерфейс

Вписуємо математичний вираз і натискаємо кнопку «Порахувати». Програма знає такі оператори: +, -, *, /, ^ (Рисунок 3.2.2)

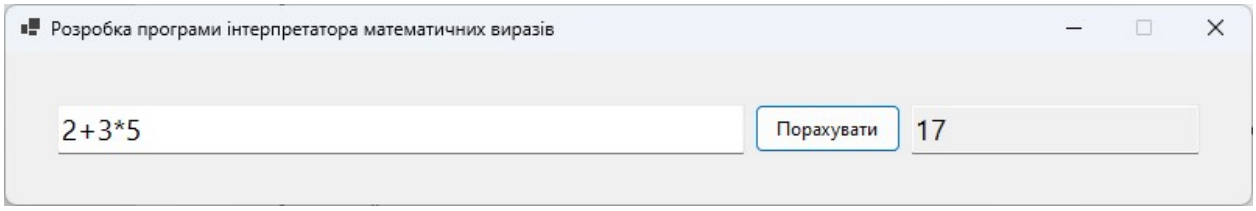

Рисунок 3.2.2 – Обчислення + *

Додаємо ще одну дію додавання (Рисунок 3.2.3)

Рисунок 3.2.3 – Обчислення + * +

Зміна математичних дій у виразі (Рисунок 3.2.4)

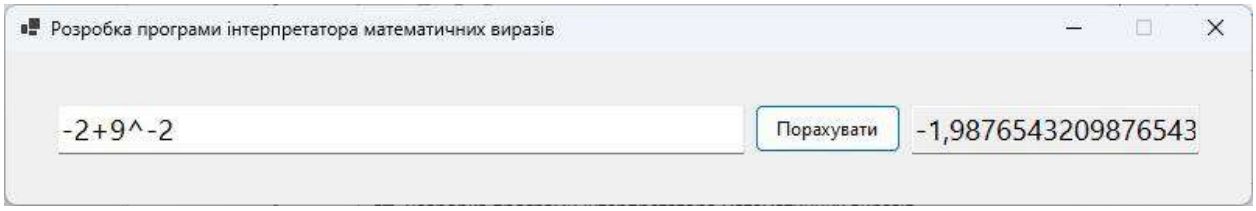

Рисунок 3.2.4 – Обчислення - + \ -

Програма знає такі функції: ABS, SIN, COS, TAN, LOG, EXP, SQRT. Кут задається в градусах.

Використання тригонометричних функцій (Рисунок 3.2.5)

Рисунок 3.2.5 – Тригонометричні функції

Застосування логарифмів у математичних виразах (Рисунок 3.2.6)

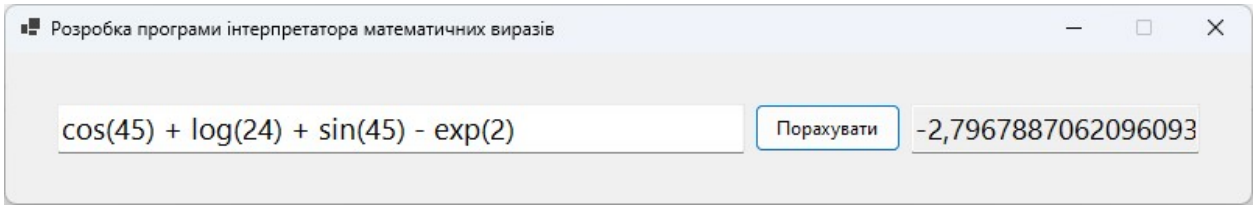

Рисунок 3.2.6 – Логарифми

Висновки до розділу 3

У даному розділі було детально розглянуто основні аспекти розробки інтерпретатора математичних виразів, включаючи вибір мови програмування, алгоритми токенізації, обчислення значень, обробки помилок та оптимізації. Вибір мови програмування C# було обґрунтовано її високою продуктивністю, потужними засобами для розробки інтерфейсів користувача, багатим набором бібліотек та зручністю роботи з об'єктно-орієнтованими концепціями. Це дозволило реалізувати ефективний та зручний інтерпретатор, який здатний обробляти складні математичні вирази.

Алгоритм токенізації був розроблений для розбиття вхідного виразу на окремі токени, що полегшило їх подальшу обробку. Було визначено важливість правильного визначення типів токенів для коректного синтаксичного аналізу виразів. Рекурсивний алгоритм обчислення значень дозволив ефективно обробляти синтаксичне дерево виразу, забезпечуючи правильне виконання арифметичних операцій та функцій.

Особлива увага була приділена обробці помилок, що забезпечує стабільну та надійну роботу програми. Було розглянуто механізми виявлення та обробки помилок на різних етапах роботи інтерпретатора, що дозволяє своєчасно інформувати користувача про можливі помилки у введених виразах.

Алгоритми оптимізації були розроблені для підвищення швидкодії та ефективності роботи інтерпретатора. Було проаналізовано методи виявлення константних підвиразів, зведення константних виразів та мінімізації кількості операцій, що дозволило значно зменшити час обчислення складних виразів.

Таким чином, проведене дослідження та розробка програми "Інтерпретатор математичних виразів" продемонстрували важливість ретельного вибору технологій та алгоритмів для досягнення високої продуктивності та надійності програмного продукту. Отримані результати підтверджують ефективність обраних підходів та технологій, що забезпечує успішну реалізацію поставлених задач.

ВИСНОВКИ

У висновках до розробки програми "Інтерпретатор математичних виразів" підсумовано основні досягнення та результати проведеної роботи. Основна задача розробки полягала в створенні ефективного та надійного інтерпретатора, здатного коректно обробляти і виконувати математичні вирази.

Проведений огляд існуючих рішень дозволив виявити основні вимоги до програми та обрати найбільш підходящі технології. Вибір мови програмування C# був обґрунтований її високою продуктивністю, зручністю роботи з об'єктно-орієнтованими концепціями, потужними інструментами для розробки графічних інтерфейсів та наявністю широкого набору бібліотек для математичних обчислень.

Розробка алгоритмів токенізації, обчислення значень, обробки помилок та оптимізації дозволила забезпечити високу точність і швидкість обробки математичних виразів. Токенізація виразів забезпечила розбиття вхідного виразу на окремі токени, що значно полегшило їх подальшу обробку. Рекурсивний спуск дозволив ефективно обробляти синтаксичне дерево виразу, забезпечуючи правильне виконання арифметичних операцій та функцій. Особлива увага була приділена обробці помилок, що забезпечило стабільну і надійну роботу програми, своєчасно інформуючи користувача про можливі помилки у введених виразах. Оптимізація обчислень сприяла підвищенню швидкодії програми та ефективності використання ресурсів.

В результаті розробки було створено інтерпретатор математичних виразів, який відповідає поставленим вимогам і завданням. Програма здатна коректно обробляти різні види математичних виразів, включаючи складні вирази з використанням функцій і операторів. Проведене тестування підтвердило надійність і точність обчислень, що дозволяє рекомендувати програму для практичного використання.

Таким чином, реалізований інтерпретатор математичних виразів демонструє ефективність обраних технологій і підходів, забезпечуючи високу продуктивність, точність та зручність використання. Проведена робота підтвердила можливість створення надійного та ефективного інтерпретатора, який може стати корисним інструментом для широкого кола користувачів.

Дипломна робота на тему "Розробка програми 'Інтерпретатор математичних виразів'" присвячена створенню програмного продукту, який дозволяє користувачам виконувати математичні обчислення за допомогою введення виразів у текстовій формі. Метою цієї роботи є аналіз існуючих алгоритмів обробки математичних виразів, вибір найбільш оптимальних підходів та їх реалізація з використанням сучасних технологій програмування на мові C#.

Робота складається з декількох розділів, кожен з яких охоплює певний аспект розробки програми. У вступній частині розглядається актуальність теми, формулюються мета та завдання дослідження. Наступні розділи присвячені огляду аналогічних програмних продуктів, обґрунтуванню вибору мови програмування та технологій, опису алгоритмів, які використовуються для парсингу, обчислення та обробки математичних виразів.

Одним з ключових етапів роботи є аналіз і вибір алгоритмів, таких як токенізація, рекурсивний спуск, обробка помилок та оптимізація. Вибір цих алгоритмів обґрунтований їх ефективністю та здатністю забезпечити коректне та швидке виконання обчислень. Алгоритм токенізації дозволяє розбити вираз на окремі елементи (токени), такі як числа, оператори, функції та дужки. Рекурсивний спуск забезпечує обробку синтаксису виразів та побудову дерева обчислень. Обробка помилок гарантує коректну роботу програми в умовах некоректного вводу, а оптимізація сприяє підвищенню швидкості та зменшенню використання ресурсів.

У розділі, присвяченому обґрунтуванню вибору технологій, детально розглядаються переваги мови C# для даного проекту. Зокрема, зазначається висока продуктивність, зручність роботи з об'єктно-орієнтованими

конструкціями, розвинуте середовище розробки Visual Studio, а також наявність широкої бази бібліотек та інструментів для роботи з математичними виразами.

Результатом роботи є готовий програмний продукт, який дозволяє користувачам вводити та обчислювати математичні вирази, включаючи підтримку різноманітних функцій, операторів та складних виразів. Програма була протестована на різних прикладах для забезпечення її коректності та надійності.

Висновки дипломної роботи підсумовують досягнуті результати, окреслюють можливі напрямки подальшого вдосконалення програми та її застосування в різних галузях, таких як освіта, наука, інженерія та фінанси. Завдяки реалізованому інтерпретатору математичних виразів користувачі отримали зручний та ефективний інструмент для виконання складних обчислень, що значно спрощує їхню роботу та підвищує продуктивність.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dotnet/Csharp-Tutorials. GitHub. URL: https://github.com/dotnet/csharp-tutorials (дата звернення: 06.06.2024).
2. C# Programming Yellow Book. Rob Miles. URL: http://www.robmiles.com/c-yellow-book/ (дата звернення: 06.06.2024).
3. Roslyn .NET Compiler Platform SDK. Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/ (дата звернення: 06.06.2024).
4. Advanced C# - Programming Techniques. Udemy. URL: https://www.udemy.com/course/advanced-csharp/ (дата звернення: 06.06.2024).
5. .NET API Browser. Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/api/ (дата звернення: 06.06.2024).
6. Understanding and Implementing Interfaces in C#. Pluralsight. URL: https://www.pluralsight.com/courses/understanding-implementing-interfaces-csharp (дата звернення: 06.06.2024).
7. C# Corner. Online Community for Developers. URL: https://www.c-sharpcorner.com/ (дата звернення: 06.06.2024).
8. Asynchronous Programming with Async and Await. Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ (дата звернення: 06.06.2024).
9. LINQ in C#. TutorialsTeacher. URL: https://www.tutorialsteacher.com/linq (дата звернення: 06.06.2024).
10. C# and .NET Online Courses. Coursera. URL: https://www.coursera.org/courses?query=c%23 (дата звернення: 06.06.2024).

11. Object-Oriented Programming in C#. Pluralsight. URL: https://www.pluralsight.com/courses/csharp-object-oriented-programming (дата звернення: 06.06.2024).
12. Data Structures and Algorithms in C#. GeeksforGeeks. URL: https://www.geeksforgeeks.org/data-structures-and-algorithms-in-c-sharp/ (дата звернення: 06.06.2024).
13. C# for Beginners. Codecademy. URL: https://www.codecademy.com/learn/learn-c-sharp (дата звернення: 06.06.2024).
14. Regular Expressions in C#. Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference (дата звернення: 06.06.2024).
15. Mastering C# and .NET Framework. Udemy. URL: https://www.udemy.com/course/mastering-csharp-and-dotnet-framework/ (дата звернення: 06.06.2024).
16. C# Performance Optimization. Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/standard/optimization/ (дата звернення: 06.06.2024).
17. Multi-threading and Parallel Programming in C#. Pluralsight. URL: https://www.pluralsight.com/courses/csharp-multithreading-parallel-programming (дата звернення: 06.06.2024).
18. Serialization in C#. Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/standard/serialization/ (дата звернення: 06.06.2024).
19. Modern C# Development with .NET. LinkedIn Learning. URL: https://www.linkedin.com/learning/modern-c-sharp-development-with-dot-net (дата звернення: 06.06.2024).
20. Programming C# 8.0. O'Reilly Media. URL: https://www.oreilly.com/library/view/programming-c-80/9781492056817/ (дата звернення: 06.06.2024).

ДОДАТКИ

ДОДАТОК А.

Програмний код модулів

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace MathInterpreter
{
   // Останній варіант класу Parser, який підтримує функції
   public class Parser
   {
      private readonly List<Token> _tokens;
      private int _currentTokenIndex;
      public Parser(List<Token> tokens)
      {
         _tokens = tokens;
         _currentTokenIndex = 0;
      }
      public Node Parse()
      {
         return Expression();
      }
      private Node Expression()
      {
         Node left = Term();
         while (_currentTokenIndex < _tokens.Count &&
               (_tokens[_currentTokenIndex].Type == TokenType.Operator &&
                (_tokens[_currentTokenIndex].Value == "+" ||
_tokens[_currentTokenIndex].Value == "-")))
         {
            Token token = _tokens[_currentTokenIndex];
            _currentTokenIndex++;
```

```
            Node right = Term();
            left = new OperationNode(char.Parse(token.Value), left, right);
        }
        return left;
    }
    private Node Term()
    {
        Node left = Power();
        while (_currentTokenIndex < _tokens.Count &&
            (_tokens[_currentTokenIndex].Type == TokenType.Operator &&
            (_tokens[_currentTokenIndex].Value == "*" ||
            _tokens[_currentTokenIndex].Value == "/")))
        {
            Token token = _tokens[_currentTokenIndex];
            _currentTokenIndex++;
            Node right = Power();
            if (token.Value == "*")
            {
                left = new OperationNode('*', left, right);
            }
            else if (token.Value == "/")
            {
                left = new OperationNode('/', left, right);
            }
            else
            {
                left = new OperationNode(char.Parse(token.Value), left, right);
            }
        }
        return left;
    }
    private Node Power()
    {
        Node left = Factor();
        while (_currentTokenIndex < _tokens.Count &&
```

```
                (_tokens[_currentTokenIndex].Type == TokenType.Operator &&
                 (_tokens[_currentTokenIndex].Value == "^")))
            {
                throw new InvalidOperationException("Невідомий оператор");
        }
        private Node Factor()
        {
            Token token = _tokens[_currentTokenIndex];
            _currentTokenIndex++;
            if (token.Type == TokenType.Number)
            {
                return new NumberNode(double.Parse(token.Value));
            }
            else if (token.Type == TokenType.Function)
            {
                if (_tokens[_currentTokenIndex].Type != TokenType.LeftParenthesis)
                {
                    throw new InvalidOperationException("Очікується ліва дужка після
імені функції");
                }
                _currentTokenIndex++; // Пропуск лівої дужки
                Node argument = Expression(); // Параметр функції
                if (_tokens[_currentTokenIndex].Type != TokenType.RightParenthesis)
                {
                    throw new InvalidOperationException("Очікується права дужка
після аргумента функції");
                }
                _currentTokenIndex++; // Пропуск правої дужки
                return new FunctionNode(token.Value, argument);
            }
            else if (token.Type == TokenType.LeftParenthesis)
            {
                Node expression = Expression();
                if (_tokens[_currentTokenIndex].Type != TokenType.RightParenthesis)
                {
```

```
            throw new InvalidOperationException("Непарна дужка");
        }
        _currentTokenIndex++; // Пропуск правої дужки
        return expression;
    }
    else if (token.Type == TokenType.Operator && token.Value == "-")
    {
        Node operand = Factor();
        // Побудова вузла унарного мінуса
        return new OperationNode('-', new NumberNode(0), operand);
    }
    else if (token.Type == TokenType.Operator && token.Value == "^")
    {
        Node left = Factor();
        Node right = Factor();
        // Побудова вузла степеню
        return new OperationNode('^', left, right);
    }
    else
    {
        throw new InvalidOperationException("Невідомий токен");
         /// </summary>
         private System.ComponentModel.IContainer components = null;
         /// <summary>
         ///  Clean up any resources being used.
         /// </summary>
         /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
         protected override void Dispose(bool disposing)
         {
            if (disposing && (components != null))
            {
                components.Dispose();
         /// <summary>
         ///  Required method for Designer support - do not modify
```

```
        ///  the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            System.ComponentModel.ComponentResourceManager resources =
new System.ComponentModel.ComponentResourceManager(typeof(Form1));
            button1 = new Button();
            textBox1 = new TextBox();
            tableLayoutPanel1 = new TableLayoutPanel();
            textBox4 = new TextBox();
            textBoxResult = new TextBox();
            textBox3 = new TextBox();
            tableLayoutPanel1.SuspendLayout();
            button1.Dock = DockStyle.Fill;
            button1.Location = new Point(451, 3);
            button1.Name = "button1";
            button1.Size = new Size(94, 32);
            button1.TabIndex = 0;
            button1.Text = "Порахувати";
            button1.UseVisualStyleBackColor = true;
            button1.Click += button1_Click;
            textBox1.Font = new Font("Segoe UI", 14F);
            textBox1.Location = new Point(3, 3);
            textBox1.Name = "textBox1";
            textBox1.Size = new Size(442, 32);
            textBox1.TabIndex = 1;
            tableLayoutPanel1.ColumnCount = 3;
            tableLayoutPanel1.ColumnStyles.Add(new
ColumnStyle(SizeType.Percent, 70F));
            tableLayoutPanel1.ColumnStyles.Add(new
ColumnStyle(SizeType.Absolute, 100F));
            tableLayoutPanel1.ColumnStyles.Add(new
ColumnStyle(SizeType.Percent, 30F));
            tableLayoutPanel1.Controls.Add(textBox4, 0, 1);
            tableLayoutPanel1.Controls.Add(textBoxResult, 2, 0);
```

```
            tableLayoutPanel1.Controls.Add(textBox1, 0, 0);
            tableLayoutPanel1.Controls.Add(button1, 1, 0);
            tableLayoutPanel1.Controls.Add(textBox3, 0, 1);
            tableLayoutPanel1.Dock = DockStyle.Fill;
            tableLayoutPanel1.Location = new Point(30, 30);
            tableLayoutPanel1.Name = "tableLayoutPanel1";
            tableLayoutPanel1.RowCount = 2;
            tableLayoutPanel1.RowStyles.Add(new RowStyle());
            tableLayoutPanel1.RowStyles.Add(new
RowStyle(SizeType.Absolute, 150F));
            tableLayoutPanel1.Size = new Size(740, 279);
            tableLayoutPanel1.SetColumnSpan(textBox4, 2);
            textBox4.Dock = DockStyle.Fill;
            textBox4.ForeColor = Color.Red;
            textBox4.Location = new Point(451, 41);
            textBox4.Multiline = true;
            textBox4.Name = "textBox4";
            textBox4.ReadOnly = true;
            textBox4.Size = new Size(286, 235);
            textBox4.TabIndex = 4;
            textBox4.Text = "Приклади використання:\r\n\t2+3*4-
4\r\n\t10^2+5/2\r\n\tcos(90)\r\n\tcos(89,5 + sin(30)) + exp(1)\r\n\tcos(2)^2 +
            textBoxResult.Dock = DockStyle.Fill;
            textBoxResult.Font = new Font("Segoe UI", 14F);
            textBoxResult.Location = new Point(551, 3);
            textBoxResult.Name = "textBoxResult";
            textBoxResult.ReadOnly = true;
            textBox3.Dock = DockStyle.Fill;
            textBox3.ForeColor = Color.Blue;
            textBox3.Location = new Point(3, 41);
            textBox3.Multiline = true;
            textBox3.Name = "textBox3";
            textBox3.ReadOnly = true;
            textBox3.Size = new Size(442, 235);
            AutoScaleDimensions = new SizeF(7F, 15F);
```

```
            AutoScaleMode = AutoScaleMode.Font;
            ClientSize = new Size(800, 339);
            Controls.Add(tableLayoutPanel1);
            FormBorderStyle = FormBorderStyle.FixedSingle;
            Icon = (Icon)resources.GetObject("$this.Icon");
            MaximizeBox = false;
            Name = "Form1";
            Padding = new Padding(30);
            Text = "Інтерпретатор математичних виразів. Автор: Закусило Олександр Олександрович";
            tableLayoutPanel1.ResumeLayout(false);
            tableLayoutPanel1.PerformLayout();
            ResumeLayout(false);
        Button button1;
            private TextBox textBox1;
            private TableLayoutPanel tableLayoutPanel1;
            private TextBox textBoxResult;
            private TextBox textBox3;
            private TextBox textBox4;
        }
}
```