|  |  |  |
| --- | --- | --- |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное автономное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | |
|  | Институт компьютерных наук и технологий | |
| **ОТЧЁТ**  по индивидуальной работе №2  по дисциплине «Язык программирования Python»  Вариант 20 | | |
|  | | Работу выполнил  студент группы ИТ-6, 2024-2025 1 курса  Булатов А. А.  «12» Июня 2025 г. |
| Работу проверил  Рубцова М. Б.  «12-14» Июня 2024 г. |
| Пермь 2025 | | |

СОДЕРЖАНИЕ

[Постановка задачи 3](#_Toc200569096)

[Алгоритм решения 3](#_Toc200569097)

[Тестирование 5](#_Toc200569098)

[Код программы 8](#_Toc200569099)

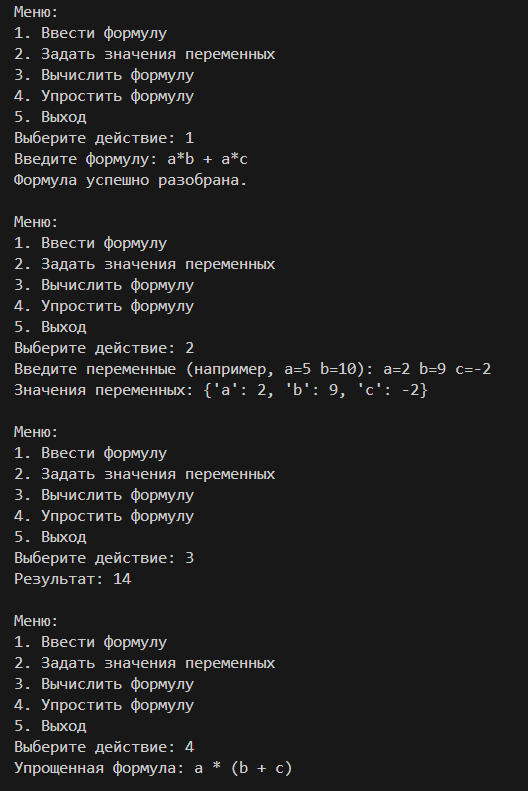
# Постановка задачи

Написать программу, которая по заданной формуле строит дерево и производит вычисления с помощью построенного дерева. Формула задана в традиционной инфиксной записи, в ней могут быть скобки, максимальная степень вложенности которых ограничивается числом 10. Аргументами могут быть целые числа и переменные, задаваемые однобуквенными именами. Допустимые операции: +, -, \*, /. Унарный минус допустим. С помощью построенного дерева формулы упростить формулу, заменяя в ней все поддеревья, соответствующие формулам (f1\*f3±f2\*f3) и (f1\*f2±f1\*f3) на поддеревья, соответствующие формулам ((f1±f2)\*f3) и (f1\*(f2±f3)).

# Алгоритм решения

1. **Построение дерева выражений**
   * Токеницация:
     + Вход: строка выражения (например, "ab + ac")
     + Выход: список токенов (['a', '', 'b', '+', 'a', '', 'c'])
     + Особенности:
       1. Поддержка унарного минуса
       2. Проверка допустимых символов
   * Преобразование в постфиксную запись (алгоритм сортировочной станции):
     + Обработка приоритетов операторов
     + Учет скобок
   * Построение дерева:
     + Стековая машина для создания узлов
     + Особенности:
       1. Числа/переменные → листья
       2. Операторы → узлы с потомками
2. **Вычисление выражения**
   * Рекурсивный обход дерева:
     + Листья:
       1. Числа → возврат значения
       2. Переменные → подстановка из словаря
     + Узлы-операторы:
       1. Вычисление левого и правого поддеревьев
       2. Применение операции
   * Обработка ошибок:
     + Неопределенные переменные
     + Деление на ноль
3. **Упрощение выражения**
   * Шаблоны упрощения:
     + (f1\*f3 ± f2\*f3) → (f1 ± f2)\*f3
     + (f1\*f2 ± f1\*f3) → f1\*(f2 ± f3)
   * Алгоритм:
     + Рекурсивный обход дерева
     + Поиск совпадающих поддеревьев
     + Перестройка узлов при обнаружении шаблона
   * Проверка эквивалентности поддеревьев:
     + Рекурсивное сравнение узлов
4. **Преобразование дерева в строку**
   * Рекурсивный обход с учетом:
     + Приоритетов операторов
     + Расстановки скобок
5. **Обработка ошибок**
   * Синтаксические:
     + Несбалансированные скобки
     + Недопустимые символы
   * Вычислительные:
     + Неопределенные переменные
     + Деление на ноль
   * Логические:
     + Пустые выражения
     + Неверный формат ввод

# Тестирование

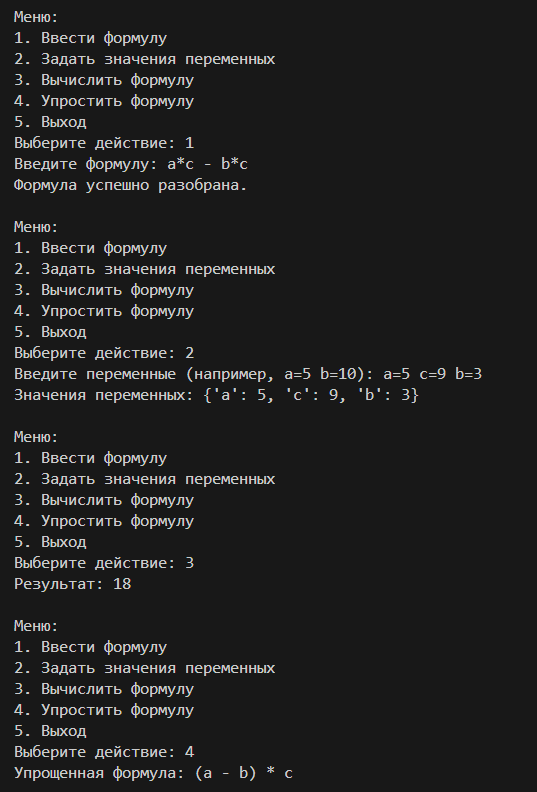


a\*b + a\*c

A=2 b=9 c=(-2)

2\*9 + 2\*(-2) = 18 – 4 = 14

2 \* (9 - 2) = 2\*7 = 14

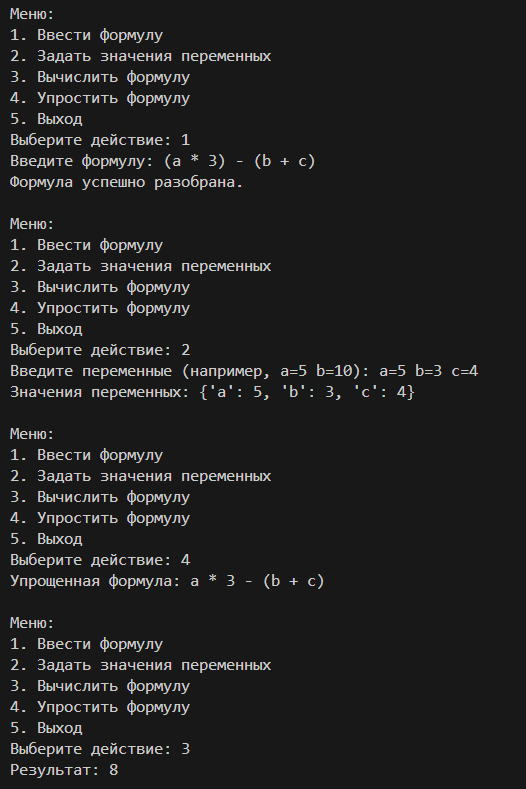


a\*с – b\*c

a=5 c=9 b=3

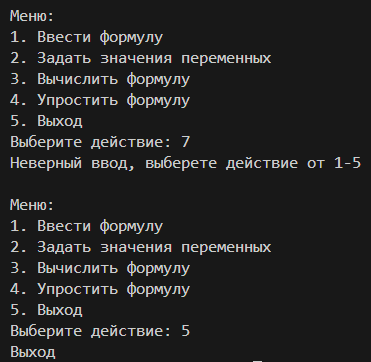
5\*9 – 3\*9 = 45 – 27 = 18

(5-3) \* 9 = 2\*9 = 18



(A\*3) – (b + c)

15 – 7 = 8



# Код программы

class TreeNode:   # Класс узла дерева выражений

    def \_\_init\_\_(self, value):

        self.value = value  # Значение узла (число, переменная или оператор)

        self.left = None

        self.right = None

    def \_\_str\_\_(self):

        return str(self.value)

class ExpressionTree:   # Класс для работы с деревом выражений

    def \_\_init\_\_(self):

        self.root = None  # Корень дерева

    def build\_from\_infix(self, expression):   # Построение дерева из инфиксной записи

        tokens = self.tokenize(expression)

        postfix = self.infix\_to\_postfix(tokens)

        self.root = self.build\_tree(postfix)

        return self.root

    def tokenize(self, expression):    # Разбиение выражения на токены

        tokens = []

        i = 0

        n = len(expression)

        while i < n:

            if expression[i] == ' ':

                i += 1

                continue

if i > 0 and (expression[i].isalpha() and expression[i-1].isdigit()) or \

                         (expression[i].isalpha() and expression[i-1].isalpha()):

                raise ValueError(

                    "Ошибка: неявное умножение. Используйте явный знак '\*' (например '3\*b' вместо '3b')")

            if expression[i] in '()+-\*/':

                # Обработка унарного минуса

                if expression[i] == '-' and (not tokens or tokens[-1] == '(' or tokens[-1] in '+-\*/'):

                    tokens.append('u-')  # Специальный маркер для унарного минуса

                    i += 1

                else:

                    tokens.append(expression[i])

                    i += 1

            elif expression[i].isdigit():

                j = i

                while j < n and expression[j].isdigit():

                    j += 1

                tokens.append(expression[i:j])

                i = j

            elif expression[i].isalpha():

                tokens.append(expression[i])

                i += 1

            else:

                raise ValueError(f"Недопустимый символ: {expression[i]}")

        return tokens

    def infix\_to\_postfix(self, tokens):   # Преобразование инфиксной записи в постфиксную

        output = []

        stack = []

        precedence = {'+': 1, '-': 1, '\*': 2, '/': 2, 'u-': 3}

        for token in tokens:

            if token.isdigit() or token.isalpha():

                output.append(token)

            elif token == '(':

                stack.append(token)

            elif token == ')':

                while stack and stack[-1] != '(':

                    output.append(stack.pop())

                stack.pop()  # Удаляем '('

            else:  # оператор или унарный минус

                while (stack and stack[-1] != '(' and

                       precedence.get(stack[-1], 0) >= precedence.get(token, 0)):

                    output.append(stack.pop())

                stack.append(token)

        while stack:

            output.append(stack.pop())

        return output

    def build\_tree(self, postfix):   # Построение дерева из постфиксной записи

        stack = []

        for token in postfix:

            if token in '+-\*/':

                node = TreeNode(token)

                node.right = stack.pop()

                node.left = stack.pop()

                stack.append(node)

            elif token == 'u-':

                node = TreeNode('-')

                node.right = stack.pop()

                stack.append(node)

            else:

                stack.append(TreeNode(token))

        if not stack:

            raise ValueError("Неверное выражение")

        return stack[0]

    def to\_expression(self, node=None):   # Преобразование дерева обратно в строку

        if node is None:

            node = self.root

        if not node:

            return ""

        if not node.left and not node.right:

            return str(node.value)

        left\_str = self.to\_expression(node.left)

        right\_str = self.to\_expression(node.right)

        # Добавляем скобки по приоритетам

        if node.left and node.left.value in '+-\*/' and self.get\_precedence(node.left.value) < self.get\_precedence(node.value):

            left\_str = f"({left\_str})"

        if node.right and node.right.value in '+-\*/' and self.get\_precedence(node.right.value) <= self.get\_precedence(node.value):

            right\_str = f"({right\_str})"

        return f"{left\_str} {node.value} {right\_str}"

    def get\_precedence(self, op):   # Получение приоритета оператора

        if op in '\*/':

            return 2

        elif op in '+-':

            return 1

        return 0

class FormulaProcessor:   # Класс для вычисления и упрощения формул

    def \_\_init\_\_(self):

        self.tree = ExpressionTree()

        self.variables = {}

    def set\_formula(self, expression):   # Установка формулы

        self.tree.build\_from\_infix(expression)

    def set\_variable(self, name, value):   # Установка значения переменной

        self.variables[name] = value

    def evaluate(self, node=None):   # Вычисление значения формулы

        if node is None:

            node = self.tree.root

            if node is None:

                raise ValueError("Формула не задана")

        if node.value.isdigit():

            return int(node.value)

        elif node.value.startswith('-') and node.value[1:].isdigit():

            return -int(node.value[1:])

        elif node.value.isalpha():

            if node.value not in self.variables:

                raise ValueError(f"Неопределённая переменная: {node.value}")

            return self.variables[node.value]

        left\_val = self.evaluate(node.left) if node.left else 0

        right\_val = self.evaluate(node.right) if node.right else 0

        if node.value == '+':

            return left\_val + right\_val

        elif node.value == '-':

            return left\_val - right\_val

        elif node.value == '\*':

            return left\_val \* right\_val

        elif node.value == '/':

            if right\_val == 0:

                raise ValueError("Деление на ноль")

            return left\_val / right\_val

        else:

            raise ValueError(f"Неизвестный оператор: {node.value}")

    def simplify(self):   # Упрощение формулы

        if self.tree.root is None:

            raise ValueError("Формула не задана")

        self.tree.root = self.simplify\_node(self.tree.root)

        return self.tree.to\_expression()

    def simplify\_node(self, node):   # Рекурсивное упрощение узла

        if not node:

            return node

        node.left = self.simplify\_node(node.left)

        node.right = self.simplify\_node(node.right)

        if node.value in '+-':

            node = self.try\_simplify\_pattern(node)

        return node

    def try\_simplify\_pattern(self, node):   # Попытка упрощения по заданным шаблонам

        left = node.left

        right = node.right

        if left and left.value == '\*' and right and right.value == '\*':

            # Шаблоны: (f1\*f3 +- f2\*f3) -> (f1 +- f2)\*f3

            #          (f1\*f2 +- f1\*f3) -> f1\*(f2 +- f3)

            ll, lr = left.left, left.right

            rl, rr = right.left, right.right

            # Проверяем 4 возможных варианта совпадения множителей

            for case in [(lr, rr), (ll, rl), (ll, rr), (lr, rl)]:

                if self.is\_equal\_trees(\*case):

                    a, b = case

                    new\_node = TreeNode('\*')

                    if a == lr and b == rr:  # (f1\*f3 +- f2\*f3)

                        new\_left = TreeNode(node.value)

                        new\_left.left = ll

                        new\_left.right = rl

                        new\_node.left = new\_left

                        new\_node.right = lr

                    elif a == ll and b == rl:  # (f1\*f2 +- f1\*f3)

                        new\_right = TreeNode(node.value)

                        new\_right.left = lr

                        new\_right.right = rr

                        new\_node.left = ll

                        new\_node.right = new\_right

                    return new\_node

        return node

    def is\_equal\_trees(self, tree1, tree2):   # Проверка равенства двух поддеревьев

        if not tree1 and not tree2:

            return True

        if not tree1 or not tree2:

            return False

        return (tree1.value == tree2.value and

                self.is\_equal\_trees(tree1.left, tree2.left) and

                self.is\_equal\_trees(tree1.right, tree2.right))

def main():   # Основная функция программы

    print("Программа для работы с формулами")

    processor = FormulaProcessor()

    while True:

        print("\nМеню:")

        print("1. Ввести формулу")

        print("2. Задать значения переменных")

        print("3. Вычислить формулу")

        print("4. Упростить формулу")

        print("5. Выход")

        choice = input("Выберите действие: ").strip()

        if choice == '1':

            try:

                expr = input("Введите формулу: ").strip()

                processor.set\_formula(expr)

                print("Формула успешно разобрана.")

            except Exception as e:

                print(f"Ошибка: {e}")

        elif choice == '2':

            try:

                var\_input = input("Введите переменные (например, a=5 b=10): ").strip()

                if var\_input:

                    for part in var\_input.split():

                        name, value = part.split('=')

                        processor.set\_variable(name, int(value))

                print("Значения переменных:", processor.variables)

            except Exception as e:

                print(f"Ошибка: {e}")

        elif choice == '3':

            try:

                result = processor.evaluate()

                print(f"Результат: {result}")

            except Exception as e:

                print(f"Ошибка вычисления: {e}")

        elif choice == '4':

            try:

                simplified = processor.simplify()

                print(f"Упрощенная формула: {simplified}")

            except Exception as e:

                print(f"Ошибка упрощения: {e}")

        elif choice == '5':

            print("Выход")

            break

        else:

            print("Неверный ввод, выберете действие от 1-5")

if \_\_name\_\_ == "\_\_main\_\_":

    main()