**Отчет по лабораторной работе № 24**

**по практикуму по циклу дисциплин "Информатика"**

Выполнил студент группы М8О-114БВ-24: Дробышев Егор Павлович, № по списку 9

Контакты e-mail: [tru.899@yandex.ru](mailto:tru.899@yandex.ru)

Работа выполнена: «12» апреля 2025 г.

Преподаватель: доцент каф. 806 Никулин Сергей Петрович

Отчет сдан « » 2025г., итоговая оценка

Подпись преподавателя

1. **Тема:** «Преобразование выражений»
2. **Цель работы:** составить программу выполнения заданных преобразований арифметических выражений с применением деревьев.
3. **Задание**: умножение переменной на сумму заменить на сумму произведений ( вариант – 9)

# Оборудование: оборудование ПЭВМ студента, если использовалось:

Процессор \_Intel Core i5\_ с ОП 8 Гб НМД 256 Гб. Монитор 1920x1080~60Hz. Другие устройства не использовались

# Программное обеспечение: программное обеспечение ЭВМ студента:

Операционная система семейства \_Linux\_, наименование \_Ubuntu\_ версия \_24.04

интерпретатор команд \_GNU bash\_ версия \_5.2.21(1)\_.

Редактор текстов emacs версия 29.3

Утилиты операционной системы: gcc, gdb

Местонахождение и имена файлов программ и данных на домашнем компьютере:

/home/tru\_\_

1. **Идея, метод, алгоритм решения задачи** (в формах: словесной, псевдокода, графической [блок-схема, диаграмма, рисунок, таблица] или формальные спецификации с пред- и постусловиями)
2. **Построение дерева выражения**: вводится арифметическое выражение, стек используется для хранения операндов и операторов, при обработке каждого символа строится дерево, соблюдая приоритет операций.
3. **Вывод инфиксного представления**: печать выражения в стандартной записи, с учётом скобок для правильного порядка операций.
4. **Упрощение выражений (распределение)**: используется распределительное свойство умножения относительно сложения/вычитания.
5. **Сценарий выполнения работы** [план работы, первоначальный текст программы в черновике (можно на отдельном листе) и тесты либо соображения по тестированию].

#include <ctype.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef enum { node\_number, node\_operator, node\_variable } NodeType;

typedef struct TreeNode {

    NodeType type;

    union {

        double number;

        char op;

        char variable;

    } data;

    struct TreeNode \*left;

    struct TreeNode \*right;

} TreeNode;

typedef struct {

    TreeNode \*\*data;

    int top;

    int capacity;

} Stack;

Stack \*createStack(int *capacity*) {

    Stack \*stack = malloc(sizeof(Stack));

    stack->data = malloc(*capacity* \* sizeof(TreeNode \*));

    stack->top = -1;

    stack->capacity = *capacity*;

    return stack;

}

int isEmpty(Stack \**stack*) { return *stack*->top == -1; }

void push(Stack \**stack*, TreeNode \**node*) {

    if (*stack*->top == *stack*->capacity - 1) {

        printf("Переполнение стека\n");

        exit(1);

    }

*stack*->data[++*stack*->top] = *node*;

}

TreeNode \*pop(Stack \**stack*) {

    if (isEmpty(*stack*)) {

        printf("Переполнение стека\n");

        exit(1);

    }

    return *stack*->data[*stack*->top--];

}

TreeNode \*peek(Stack \**stack*) {

    return isEmpty(*stack*) ? NULL : *stack*->data[*stack*->top];

}

TreeNode \*createNumberNode(double *value*) {

    TreeNode \*node = malloc(sizeof(TreeNode));

    node->type = node\_number;

    node->data.number = *value*;

    node->left = node->right = NULL;

    return node;

}

TreeNode \*createOperatorNode(char *op*) {

    TreeNode \*node = malloc(sizeof(TreeNode));

    node->type = node\_operator;

    node->data.op = *op*;

    node->left = node->right = NULL;

    return node;

}

TreeNode \*createVariableNode(char *var*) {

    TreeNode \*node = malloc(sizeof(TreeNode));

    node->type = node\_variable;

    node->data.variable = *var*;

    node->left = node->right = NULL;

    return node;

}

int getPriority(char *op*) {

    switch (*op*) {

        case '+': case '-': return 1;

        case '\*': case '/': return 2;

        default: return 0;

    }

}

TreeNode \*buildExpressionTree(const char \**expression*) {

    Stack \*values = createStack(strlen(*expression*));

    Stack \*operators = createStack(strlen(*expression*));

    for (int i = 0; *expression*[i]; i++) {

        if (*expression*[i] == ' ') continue;

        if (*expression*[i] == '(') {

            push(operators, createOperatorNode(*expression*[i]));

        } else if (*expression*[i] == ')') {

            while (!isEmpty(operators) && peek(operators)->data.op != '(') {

                TreeNode \*opNode = pop(operators);

                opNode->right = pop(values);

                opNode->left = pop(values);

                push(values, opNode);

            }

            pop(operators);

        } else if (isdigit(*expression*[i]) || *expression*[i] == '.') {

            double num = 0;

            int dot = 0;

            double fraction = 1.0;

            while (*expression*[i] && (isdigit(*expression*[i]) || *expression*[i] == '.')) {

                if (*expression*[i] == '.') dot = 1;

                else {

                    if (dot) {

                        fraction \*= 0.1;

                        num += (*expression*[i] - '0') \* fraction;

                    } else {

                        num = num \* 10 + (*expression*[i] - '0');

                    }

                }

                i++;

            }

            i--;

            push(values, createNumberNode(num));

        } else if (isalpha(*expression*[i])) {

            push(values, createVariableNode(*expression*[i]));

        } else {

            TreeNode \*opNode = createOperatorNode(*expression*[i]);

            while (!isEmpty(operators) && getPriority(peek(operators)->data.op) >= getPriority(*expression*[i])) {

                TreeNode \*topOp = pop(operators);

                topOp->right = pop(values);

                topOp->left = pop(values);

                push(values, topOp);

            }

            push(operators, opNode);

        }

    }

    while (!isEmpty(operators)) {

        TreeNode \*opNode = pop(operators);

        opNode->right = pop(values);

        opNode->left = pop(values);

        push(values, opNode);

    }

    TreeNode \*result = pop(values);

    free(values->data); free(values);

    free(operators->data); free(operators);

    return result;

}

bool needParenthesesLeft(TreeNode \**parent*, TreeNode \**child*) {

    return *child* && *child*->type == node\_operator &&

           getPriority(*parent*->data.op) > getPriority(*child*->data.op);

}

bool needParenthesesRight(TreeNode \**parent*, TreeNode \**child*) {

    if (!*child* || *child*->type != node\_operator) return false;

    int parentP = getPriority(*parent*->data.op);

    int childP = getPriority(*child*->data.op);

    return parentP > childP || (parentP == childP && (*parent*->data.op == '-' || *parent*->data.op == '/'));

}

void printInfix(TreeNode \**root*) {

    if (!*root*) return;

    if (*root*->left && needParenthesesLeft(*root*, *root*->left)) {

        printf("(");

        printInfix(*root*->left);

        printf(")");

    } else if (*root*->left) {

        printInfix(*root*->left);

    }

    switch (*root*->type) {

        case node\_number:

            if (*root*->data.number == (int)*root*->data.number)

                printf("%d", (int)*root*->data.number);

            else

                printf("%g", *root*->data.number);

            break;

        case node\_operator:

            printf("%c", *root*->data.op);

            break;

        case node\_variable:

            printf("%c", *root*->data.variable);

            break;

    }

    if (*root*->right && needParenthesesRight(*root*, *root*->right)) {

        printf("(");

        printInfix(*root*->right);

        printf(")");

    } else if (*root*->right) {

        printInfix(*root*->right);

    }

}

void printTree(TreeNode \**root*, int *level*) {

    if (!*root*) return;

    printTree(*root*->right, *level* + 1);

    for (int i = 0; i < *level*; i++) printf("    ");

    switch (*root*->type) {

        case node\_number:

            if (*root*->data.number == (int)*root*->data.number)

                printf("%d\n", (int)*root*->data.number);

            else

                printf("%g\n", *root*->data.number);

            break;

        case node\_operator: printf("%c\n", *root*->data.op); break;

        case node\_variable: printf("%c\n", *root*->data.variable); break;

    }

    printTree(*root*->left, *level* + 1);

}

void freeTree(TreeNode \**root*) {

    if (!*root*) return;

    freeTree(*root*->left);

    freeTree(*root*->right);

    free(*root*);

}

TreeNode \*copyTree(TreeNode \**root*) {

    if (!*root*) return NULL;

    TreeNode \*node = malloc(sizeof(TreeNode));

    node->type = *root*->type;

    node->data = *root*->data;

    node->left = copyTree(*root*->left);

    node->right = copyTree(*root*->right);

    return node;

}

TreeNode \*applyDistributiveInternal(TreeNode \**node*) {

    if (!*node*) return NULL;

*node*->left = applyDistributiveInternal(*node*->left);

*node*->right = applyDistributiveInternal(*node*->right);

    if (*node*->type == node\_operator && *node*->data.op == '\*') {

        TreeNode \*left = *node*->left;

        TreeNode \*right = *node*->right;

        if (left && (left->type == node\_variable || left->type == node\_number) &&

            right && right->type == node\_operator &&

            (right->data.op == '+' || right->data.op == '-')) {

            TreeNode \*newLeft = createOperatorNode('\*');

            newLeft->left = copyTree(left);

            newLeft->right = copyTree(right->left);

            TreeNode \*newRight = createOperatorNode('\*');

            newRight->left = copyTree(left);

            newRight->right = copyTree(right->right);

            TreeNode \*newRoot = createOperatorNode(right->data.op);

            newRoot->left = newLeft;

            newRoot->right = newRight;

            TreeNode \*result = applyDistributiveInternal(newRoot);

            freeTree(*node*);

            return result;

        }

        if (right && (right->type == node\_variable || right->type == node\_number) &&

            left && left->type == node\_operator &&

            (left->data.op == '+' || left->data.op == '-')) {

            TreeNode \*newLeft = createOperatorNode('\*');

            newLeft->left = copyTree(left->left);

            newLeft->right = copyTree(right);

            TreeNode \*newRight = createOperatorNode('\*');

            newRight->left = copyTree(left->right);

            newRight->right = copyTree(right);

            TreeNode \*newRoot = createOperatorNode(left->data.op);

            newRoot->left = newLeft;

            newRoot->right = newRight;

            TreeNode \*result = applyDistributiveInternal(newRoot);

            freeTree(*node*);

            return result;

        }

    }

    return *node*;

}

TreeNode \*applyDistributive(TreeNode \**node*) {

    return applyDistributiveInternal(copyTree(*node*));

}

void printMenu() {

    printf("\nМеню:\n1. Ввести выражение\n2. Вывести выражение\n3. Упростить выражение\n4. Выход\nВыберите пункт: ");

}

int main() {

    char expression[256] = "";

    TreeNode \*root = NULL, \*simplifiedRoot = NULL;

    int choice;

    bool hasExpression = false;

    do {

        printMenu();

        scanf("%d", &choice);

        while (getchar() != '\n');

        switch (choice) {

            case 1:

                printf("Введите арифметическое выражение: ");

                fgets(expression, sizeof(expression), stdin);

                expression[strcspn(expression, "\n")] = '\0';

                freeTree(root); root = NULL;

                freeTree(simplifiedRoot); simplifiedRoot = NULL;

                root = buildExpressionTree(expression);

                hasExpression = true;

                printf("Выражение сохранено.\n");

                break;

            case 2:

                if (!hasExpression) {

                    printf("Сначала введите выражение (пункт 1).\n");

                    break;

                }

                printf("\nТекстовое представление: ");

                printInfix(root);

                printf("\n\nДерево выражения:\n");

                printTree(root, 0);

                break;

            case 3:

                if (!hasExpression) {

                    printf("Сначала введите выражение (пункт 1).\n");

                    break;

                }

                freeTree(simplifiedRoot);

                simplifiedRoot = applyDistributive(root);

                printf("\nУпрощенное выражение (текст): ");

                printInfix(simplifiedRoot);

                printf("\n\nУпрощенное дерево выражения:\n");

                printTree(simplifiedRoot, 0);

                break;

            case 4:

                printf("Выход из программы.\n");

                break;

            default:

                printf("Неверный выбор. Попробуйте снова.\n");

        }

    } while (choice != 4);

    freeTree(root);

    freeTree(simplifiedRoot);

    return 0;

}

1. **Распечатка протокола** (подклеить листинг окончательного варианта программы с тестовыми примерами, подписанный преподавателем).

tru@empty:/mnt/c/c proj/vsc/sem2$ gcc noname.c -o noname

tru@empty:/mnt/c/c proj/vsc/sem2$ ./noname

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: a \* (b + c)

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: a\*(b+c)

Дерево выражения:

c

+

b

\*

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

Упрощенное выражение (текст): a\*b+a\*c

Упрощенное дерево выражения:

c

\*

a

+

b

\*

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: a \* (b + c + 12)

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: a\*(b+c+12)

Дерево выражения:

12

+

c

+

b

\*

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

Упрощенное выражение (текст): a\*b+a\*c+a\*12

Упрощенное дерево выражения:

12

\*

a

+

c

\*

a

+

b

\*

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: x \* (a + b) + (c + d) \* y

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: x\*(a+b)+(c+d)\*y

Дерево выражения:

y

\*

d

+

c

+

b

+

a

\*

x

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

Упрощенное выражение (текст): x\*a+x\*b+c\*y+d\*y

Упрощенное дерево выражения:

y

\*

d

+

y

\*

c

+

b

\*

x

+

a

\*

x

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 4

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: a \* (2 + b) + 5

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: a\*(2+b)+5

Дерево выражения:

5

+

b

+

2

\*

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

Упрощенное выражение (текст): a\*2+a\*b+5

Упрощенное дерево выражения:

5

+

b

\*

a

+

2

\*

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 4

Выход из программы.

tru@empty:/mnt/c/c proj/vsc/sem2$

1. **Дневник отладки** должен содержать дату и время сеансов отладки и основные события (ошибки в сценарии и программе, нестандартные ситуации) и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании других ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | Лаб. или дом. | Дата | Время | Событие | Действие по исправлению | Примечание |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

# Замечания автора по существу работы

1. **Выводы:**

В ходе выполнения лабораторной работы научился составлять программу для выполнения заданных преобразований арифметических выражений с применением деревьев. Недочёты при выполнении задания могут быть устранены следующим образом:

**Подпись студента** \_\_\_\_\_\_\_\_\_\_\_\_