**Отчет по лабораторной работе № 24**

по курсу “практикум программирования”

Студент группы: М8О-114БВ-24 Дылдин Сергей, № по списку 10

Контакты e-mail: dsergey010206@gmail.com

Работа выполнена: «06»Апреля г.

Преподаватель: доцент каф. 806 Никулин Сергей Петрович

Отчет сдан «7»Апреля 2025г., итоговая оценка\_\_\_\_\_\_\_\_

Подпись преподавателя\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Тема:** динамические структуры данных, обработка деревьев.
2. **Цель работы:** составить программу выполнения заданных преобразований арифметических выражений с применением деревьев.
3. **Задание** *вариант №10***:**



1. **Оборудование ПЭВМ студента**

Intel core i5 12450H, Xiaomi Mi silent mouse  
 Другие устройства не использовались

1. *Программное обеспечение ЭВМ студента, если использовалось:*

Операционная система семейства \_Linux\_, наименование \_Ubuntu\_ версия 20.04.6 LTS интерпретатор команд \_GNU/Linux\_ версия \_5.15.153.1\_.

Редактор текстов GNU nano версия 6.2

Утилиты операционной системы: Терминал, gcc

Прикладные системы и программы: gnu

Местонахождение и имена файлов программ и данных на домашнем компьютере /home/

1. **Идея, метод, алгоритм** решения задачи (в формах: словесной, псевдокода, графической [блок- схема, диаграмма, рисунок, таблица] или формальные спецификации с пред- и постусловиями)

Проверяем, что строка содержит корректное арифметическое выражение (скобки, операторы, числа, переменные).

1. Построение дерева (buildExpressionTree) - разбирает строку с выражением, используя два стека: для чисел/переменных и для операторов, формируя дерево с учётом приоритетов операций и скобок.
2. Вывод выражения (printInfix) - обходит дерево и выводит выражение в привычной форме, автоматически расставляя необходимые скобки.
3. Отображение структуры (printTree) - показывает дерево в графическом виде (вертикально, с отступами).
4. Упрощение (applyDistributive) - применяет дистрибутивный закон умножения относительно вычитания, преобразуя выражения типа a\*(b-c) в a\*b - a\*c.
5. Копирование и очистка (copyTree, freeTree) - создают копии деревьев и освобождают память.
6. **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 = (TreeNode \*\*) 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) {

if (isEmpty(stack)) {

return NULL;

}

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

}

TreeNode \*createNumberNode(double value) {

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

if (node == NULL) {

fprintf(stderr, "Ошибка выделения памяти\n");

exit(1);

}

node->type = node\_number;

node->data.number = value;

node->left = NULL;

node->right = NULL;

return node;

}

TreeNode \*createOperatorNode(char op) {

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

node->type = node\_operator;

node->data.op = op;

node->left = NULL;

node->right = NULL;

return node;

}

TreeNode \*createVariableNode(char var) {

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

node->type = node\_variable;

node->data.variable = var;

node->left = NULL;

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) {

if (child == NULL || parent == NULL)

return false;

if (child->type != node\_operator)

return false;

if (getPriority(parent->data.op) > getPriority(child->data.op)) {

return true;

}

return false;

}

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

if (child == NULL || parent == NULL)

return false;

if (child->type != node\_operator)

return false;

if (getPriority(parent->data.op) > getPriority(child->data.op)) {

return true;

}

if (getPriority(parent->data.op) == getPriority(child->data.op)) {

if (parent->data.op == '-' || parent->data.op == '/') {

return true;

}

}

return false;

}

void printInfix(TreeNode \*root) {

if (root == NULL) {

return;

}

if (root->left) {

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

printf("(");

printInfix(root->left);

printf(")");

} else {

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) {

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

printf("(");

printInfix(root->right);

printf(")");

} else {

printInfix(root->right);

}

}

}

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

if (root == NULL) {

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 == NULL)

return;

freeTree(root->left);

freeTree(root->right);

free(root);

}

void printMenu() {

printf("\nМеню:\n");

printf("1. Ввести выражение\n");

printf("2. Вывести выражение\n");

printf("3. Упростить выражение\n");

printf("4. Выход\n");

printf("Выберите пункт: ");

}

TreeNode \*copyTree(TreeNode \*root) {

if (root == NULL)

return NULL;

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

if (!newNode) {

fprintf(stderr, "Ошибка выделения памяти\n");

exit(1);

}

newNode->type = root->type;

switch (root->type) {

case node\_number:

newNode->data.number = root->data.number;

break;

case node\_operator:

newNode->data.op = root->data.op;

break;

case node\_variable:

newNode->data.variable = root->data.variable;

break;

}

newNode->left = copyTree(root->left);

newNode->right = copyTree(root->right);

return newNode;

}

// по заданию

TreeNode \*applyDistributiveInternal(TreeNode \*node) {

if (node == NULL) {

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;

// Случай: (число/переменная) \* (A - B)

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

right->type == node\_operator && right->data.op == '-') {

TreeNode \*copyLeft1 = copyTree(left);

TreeNode \*copyLeft2 = copyTree(left);

TreeNode \*a = copyTree(right->left);

TreeNode \*b = copyTree(right->right);

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

leftA->left = copyLeft1;

leftA->right = a;

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

leftB->left = copyLeft2;

leftB->right = b;

TreeNode \*newNode = createOperatorNode('-');

newNode->left = leftA;

newNode->right = leftB;

// Освобождаем текущий узел (\*), который уже не используется

free(node);

return newNode;

} else if (right != NULL && (right->type == node\_variable || right->type == node\_number) && left != NULL &&

left->type == node\_operator && left->data.op == '-') {

// Создаем две копии правого поддерева для использования в обеих частях выражения

TreeNode \*copyRight1 = copyTree(right);

TreeNode \*copyRight2 = copyTree(right);

TreeNode \*a = copyTree(left->left);

TreeNode \*b = copyTree(left->right);

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

aRight->left = a;

aRight->right = copyRight1;

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

bRight->left = b;

bRight->right = copyRight2;

TreeNode \*newNode = createOperatorNode('-');

newNode->left = aRight;

newNode->right = bRight;

free(node);

return newNode;

}

}

return node;

}

TreeNode \*applyDistributive(TreeNode \*node) {

if (node == NULL) {

return NULL;

}

TreeNode \*treeCopy = copyTree(node);

return applyDistributiveInternal(treeCopy);

}

int main() {

char expression[256] = "";

TreeNode \*root = NULL;

TreeNode \*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';

if (simplifiedRoot) {

freeTree(simplifiedRoot);

simplifiedRoot = NULL;

}

if (root) {

freeTree(root);

root = 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");

printf("\nДерево выражения:\n");

printTree(root, 0);

break;

case 3:

if (!hasExpression) {

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

break;

}

if (simplifiedRoot != NULL) {

freeTree(simplifiedRoot);

}

simplifiedRoot = copyTree(root);

simplifiedRoot = applyDistributive(simplifiedRoot);

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

printInfix(simplifiedRoot);

printf("\n");

printf("\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);

if (root != NULL) {

freeTree(root);

root = NULL;

}

if (simplifiedRoot != NULL) {

freeTree(simplifiedRoot);

simplifiedRoot = NULL;

}

return 0;

}

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

PS D:\apps\labi\2sem\laba3> ./main

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: x\*(a-b)

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: x\*(a-b)

Дерево выражения:

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

Упрощенное дерево выражения:

b

\*

x

-

a

\*

x

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: (a-b)\*5

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: (a-b)\*5

Дерево выражения:

5

\*

b

-

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

Упрощенное выражение (текст): a\*5-b\*5

Упрощенное дерево выражения:

5

\*

b

-

5

\*

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: 3\*(x-(y+z))

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: 3\*(x-(y+z))

Дерево выражения:

z

+

y

-

x

\*

3

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

Упрощенное выражение (текст): 3\*x-3\*(y+z)

Упрощенное дерево выражения:

z

+

y

\*

3

-

x

\*

3

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

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

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: (a+b)\*(c-d)

Дерево выражения:

d

-

c

\*

b

+

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

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

Упрощенное дерево выражения:

d

-

c

\*

b

+

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 1

Введите арифметическое выражение: (a-(b-c))\*d

Выражение сохранено.

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 2

Текстовое представление: (a-(b-c))\*d

Дерево выражения:

d

\*

c

-

b

-

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 3

Упрощенное выражение (текст): a\*d-(b-c)\*d

Упрощенное дерево выражения:

d

\*

c

-

b

-

d

\*

a

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)\*c

Упрощенное дерево выражения:

c

\*

b

+

a

Меню:

1. Ввести выражение

2. Вывести выражение

3. Упростить выражение

4. Выход

Выберите пункт: 4

Выход из программы.

PS D:\apps\labi\2sem\laba3>

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 программ

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

**10. Замечания автора по существу работы**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**11.Выводы:**

В ходе выполнения данной лабораторной работы я научился работать с двоичными деревьями и обрабатывать выражения заданным образом.

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