

## Отчёт по лабораторной работе №24

по курсу «Языки и методы программирования»

Выполнил студент группы М8О-111Б-23: Тимофеева Ирина  
Александровна № по списку 21

Работа выполнена: «12» мая 2024 г.

Преподаватель: каф. 806 Никулин Сергей Петрович

Входной контроль знаний с оценкой: \_\_\_\_\_

Отчет сдан «12» мая 2024 г.

Итоговая оценка: \_\_\_\_\_

Подпись преподавателя: \_\_\_\_\_

1. Тема: алгоритмы и структуры данных
2. Цель работы: составить программу выполнения заданных преобразований арифметических выражений с применением деревьев.
3. Задание: вариант 7 - редуцировать выражения, заменив операцию умножения переменной на целое число слагаемых:  $a*3 \rightarrow a + a + a$
4. Оборудование: Оборудование ПЭВМ студента, если использовалось:  
Процессор AMD Ryzen 5 5600H, ОП 16 ГБ, SSD 250 ГБ, мониторы 15" Full Hd Display и 27" BenQ  
BL2780T. Другие устройства: принтер Canon MG4520S, мышь Logitech g403, наушники HyperX Cloud
5. Программное обеспечение: Программное обеспечение ПЭВМ студента, если использовалось:  
Операционная система семейства Ubuntu, наименование версия VirtualBox  
Ubuntu 20.04.3  
интерпретатор команд bash версия 5.0.17. Система программирования C.  
Редактор текстов  
VI версия 8.1
6. Идея, метод, алгоритм решения задачи [в формах: словесной, псевдокода, графической (блок-схема, диаграмма, рисунок, таблица) или формальные спецификации с пред- и постусловиями]:  
Заранее продумать тесты.  
Составить программу.  
Протестировать.

7. Сценарий выполнения работы (план работы, первоначальный текст программы в черновике (можно на отдельном листе) и тесты либо соображения по тестированию).

```
// Реализовать алгоритм перевода из инфиксной записи арифметического выражения в
постфиксную
#include <stdio.h>
#include <malloc.h>
#include <ctype.h>
#include <string.h>
#define T char
#define MaxStack 36735895
#define MaxString 100

//операнды и скобки выражений
char OpenMatrix[3] = {'(', '[', '{'};
char CloseMatrix[3] = {')', ']', '}'};
char OperationMatrix[4] = {'+', '-', '*', '/'};

//формулы для тестирования программы
//char buf[MaxString] = "(140/2)-(30*(10-4))+(5*2)";
//char buf[MaxString] = "a * (5 - e) + g7 * 1 + ( b - c ) * d3";
//char buf[MaxString] = "(A+B)*(C+D)-E";
//char buf[MaxString] = "(a + 5) / 3*d";
char buf[MaxString] = "(a + 5) / (3 * d) - (c + (7 - e) * b) + 7 + 5*e";

//структура двоичного дерева
typedef struct tnode {      // узел дерева
    char oper;              // операнд (+ - * /)
    char* data;             // переменная или константа
    int count;              // число вхождений
    struct tnode* left;     // левый потомок
    struct tnode* right;    // правый потомок
    struct tnode* parent;   // родительский узел
} TreeNode;

typedef struct TNode
{
    T value;
    struct TNode *next;
} Node;

//стек операций для формирования постфиксной записи
struct TStack
{

```

```

Node *Head;
int Size;
int MaxSize;
};
struct TStack Stack;

//структура для вывода дерева в консоль
typedef struct TTrunk
{
    struct TTrunk* prev;
    char* str;
} Trunk;

//запись в стек
void Push(T item)
{
    if (Stack.Size > MaxStack)
    {
        printf("Stack overflow!");
    }
    else
    {
        Node *tmp = (Node *)malloc(sizeof(Node));
        if (tmp)
        {
            tmp->value = item;
            tmp->next = Stack.Head;
            Stack.Head = tmp;
            Stack.Size++;
        }
        else
        {
            printf("Stack overflowed!");
            Stack.Size = MaxStack + 1;
        }
    }
}

//запрос и удаление из стека
T Pop()
{
    if (Stack.Size == 0)
    {
        printf("Stack is empty!");
        return NULL;
    }
}

```

```

    }
    else
    {
        Node *node = Stack.Head;
        T item = Stack.Head->value;
        Stack.Head = Stack.Head->next;
        Stack.Size--;
        free(node);
        return item;
    }
}

```

```

void PrintStack(Node *node)
{
    while (node != NULL)
    {
        printf("%d", node->value);
        node = node->next;
    }
}

```

```

//проверка на открывающую скобку
int isOpenSymbol(char c)
{
    int i;
    for (i = 0; i < 3; i++)
    {
        if (c == OpenMatrix[i])
            return 1;
    }
    return 0;
}

```

```

//проверка на закрывающую скобку
int isCloseSymbol(char c)
{
    int i;
    for (i = 0; i < 3; i++)
    {
        if (c == CloseMatrix[i])
            return 1;
    }
    return 0;
}

```

```
//проверка на операнд + - * /
int isOperationSymbol(char c)
{
    int i;
    for (i = 0; i < 4; i++)
    {
        if (c == OperationMatrix[i])
            return 1;
    }
    return 0;
}
```

```
//проверка на число
int isNumber(char *str)
{
    if (str == NULL || strlen(str) == 0)
        return 0;
    int i = 0;
    while (str[i] != '\0')
        if (!isdigit(str[i++]))
            return 0;
    return 1;
}
```

```
//проверка на переменную
int isVariable(char *str)
{
    if (str == NULL || strlen(str) == 0 || !isalpha(str[0]))
        return 0;
    int i = 0;
    while (str[i] != '\0')
    {
        if (!isalpha(str[i]) && !isdigit(str[i]))
            return 0;
        i++;
    }
    return 1;
}
```

```
//приоритет операции
int PRIOR(char a)
{
    switch(a)
    {
        case '*':
```

```

    case '/':
        return 3;

    case '-':
    case '+':
        return 2;

    case '(':
        return 1;
    }
}

// перевод из инфиксной записи арифметического выражения в постфиксную
char *Postfix(char *str, int str_len)
{
    char *PostfixString = (char *)malloc(sizeof(char) * str_len);
    strcpy(PostfixString, "\0");
    int i;
    for (i = 0; i < strlen(str); i++)
    {
        char el[20] = "";
        int j = 0;
        while (str[i] != ' ' && str[i] != '\0')
        {
            char tmp = str[i];

            if (isOpenSymbol(tmp))
            {
                Push(tmp);
                break;
            }
            else if (isOperationSymbol(tmp))
            {
                while (Stack.Size > 0 && PRIOR(Stack.Head->value) >= PRIOR(tmp))
                {
                    char oper = Pop();
                    char tmp_str[2];
                    tmp_str[0] = oper;
                    tmp_str[1] = 0;
                    strcat(PostfixString, tmp_str);
                    strcat(PostfixString, " ");
                }
                Push(tmp);
                break;
            }
        }
    }
}

```

```

else if (isCloseSymbol(tmp))
{
    if (isNumber(el) || isVariable(el))
    {
        strcat(PostfixString, el);
        strcat(PostfixString, " ");
        el[0] = '\0';
    }
    char oper;
    do
    {
        oper = Pop();
        if (oper != NULL && !isOpenSymbol(oper))
        {
            char tmp_str[2];
            tmp_str[0] = oper;
            tmp_str[1] = 0;
            strcat(PostfixString, tmp_str);
            strcat(PostfixString, " ");
        }
    } while (!isOpenSymbol(oper));

    if (!isOpenSymbol(oper))
    {
        return NULL;
    }

    j--;
}
else
{
    el[j] = tmp;
}

j++;
i++;
}
el[j] = '\0';
if (isNumber(el) || isVariable(el))
{
    strcat(PostfixString, el);
    strcat(PostfixString, " ");
}
}
while (Stack.Size > 0)

```

```

{
    char oper = Pop();
    char tmp_str[2];
    tmp_str[0] = oper;
    tmp_str[1] = 0;
    strcat(PostfixString, tmp_str);
    strcat(PostfixString, " ");
}
return PostfixString;
}

```

//разделение постфиксной записи на 2 части для формирования двоичного дерева

```
int SplitPostfix(char *str)
```

```

{
    int indx = strlen(str);
    int form = 1;
    int symb;
    int oper;
    char el;
    while (indx > 0 && form > 0)
    {
        el = str[indx-1];
        while (el == ' ')
        {
            indx--;
            el = str[indx-1];
        }

        symb = 0;
        oper = 0;
        while (indx > 0 && (isalpha(el) || isdigit(el)))
        {
            symb++;
            indx--;
            el = str[indx-1];
        }
        while (indx > 0 && isOperationSymbol(el))
        {
            oper++;
            indx--;
            el = str[indx-1];
        }

        if (symb > 0)
            form--;
    }
}

```



```

        if (oper > 0)
            form++;

        if (indx > 0 && form > 0) indx--;
    }
    return indx;
}

```

//удаление лишних пробелов

```

char* Trim(char* str)
{
    int indx = strlen(str);
    char* s = (char*)malloc(sizeof(char) * indx);
    strcpy(s, str);
    while (s[indx-1] == ' ')
    {
        s[indx-1] = '\0';
        indx--;
    }
    return s;
}

```

//формирования дерева решений из постфиксной записи арифметической формулы

```

TreeNode* MakeSolutionTree(char* formula)
{
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    node->parent = NULL;
    node->left = NULL;
    node->right = NULL;
    node->data = formula;
    node->count = 0;
    node->oper = ' ';

    int len = strlen(formula);
    char oper;
    char* f = (char *)malloc(sizeof(char) * len);
    char* s1 = (char *)malloc(sizeof(char) * len);
    char* s2 = (char *)malloc(sizeof(char) * len);

    strcpy(f, formula);

    while (f[len-1] == ' ') len--;
    oper = f[len-1];
    f[len-1] = 0;
}

```

```

int split_indx = SplitPostfix(f);
    if (split_indx>0 && isOperationSymbol(oper))
    {
        strncpy(s1, f, split_indx);
        s1[split_indx-1] = '\0';
        s1 = Trim(s1);

        strcpy(s2, f+split_indx);
        s2 = Trim(s2);

        int k1 = 0;
            int k2 = 0;
        node->oper = oper;
        node->left = MakeSolutionTree(s1);
        if (node->left != NULL)
        {
            k1 = node->left->count+1;
            node->left->parent = node;
        }
        node->right = MakeSolutionTree(s2);
        if (node->right != NULL)
        {
            k2 = node->right->count+1;
            node->right->parent = node;
        }
        node->count = k1 > k2 ? k1 : k2;
    }

    return node;
}

// Функция удаления поддерева
void freemem(TreeNode* tree) {
    if (tree != NULL) {
        freemem(tree->left);
        freemem(tree->right);
        free(tree);
    }
}

//редуцирование операции умножения на сумму операций сложения
void MultiplyReduceTree(TreeNode* wood)
{
    if (wood != NULL)

```

```

{
    if (wood->count == 1 && wood->oper == '*')
    {
        if ((isNumber(wood->right->data) && isVariable(wood->left->data)) ||
(isVariable(wood->right->data) && isNumber(wood->left->data)))
        {
            int num = isNumber(wood->right->data) ? atoi(wood->right-
>data) : atoi(wood->left->data);
            if (num == 0)
            {
                wood->data = "0\0";
                wood->oper = ' ';
                wood->count = 0;
                free(wood->left);
                free(wood->right);
                wood->left = NULL;
                wood->right = NULL;
            }
            else if (num == 1)
            {
                wood->oper = ' ';
                if (isNumber(wood->right->data))
                    wood->data = wood->left->data;
                else
                    wood->data = wood->right->data;
                wood->count = 0;
                free(wood->left);
                free(wood->right);
                wood->left = NULL;
                wood->right = NULL;
            }
            else if (num > 1)
            {
                char* str = (char *)malloc(sizeof(char) * 20);
                sprintf(str, "%d", num-1);

                TreeNode* oper_node =
(TreeNode*)malloc(sizeof(TreeNode));
                if (num < 4)
                    oper_node->oper = '+';
                else
                    oper_node->oper = wood->oper;
                oper_node->count = 1;
                oper_node->parent = wood;
            }
        }
    }
}

```

```

        TreeNode* node1 =
(TreeNode*)malloc(sizeof(TreeNode));
        node1->oper = ' ';
        node1->count = 0;
        node1->parent = oper_node;

        TreeNode* node2 =
(TreeNode*)malloc(sizeof(TreeNode));
        node2->data = Trim(str);
        node2->oper = ' ';
        node2->count = 0;
        node2->parent = oper_node;

        if (isNumber(wood->right->data))
        {
            node1->data = wood->left->data;
            if (num < 4) node2->data = wood->left->data;
            wood->right = oper_node;
            oper_node->right = node2;
            oper_node->left = node1;
        }
        else
        {
            node1->data = wood->right->data;
            if (num < 4) node2->data = wood->right->data;
            wood->left = oper_node;
            oper_node->right = node1;
            oper_node->left = node2;
        }
        wood->oper = '+';
        wood->count = num-1;
    }

}

}
MultiplyReduceTree(wood->left);
MultiplyReduceTree(wood->right);
}
return;
}

```

//вспомогательная функция печати веток двоичного дерева

```
void showTrunks(Trunk *p)
```

```
{
    if (p == NULL) {
        return;
    }

```

```

    }

    showTrunks(p->prev);
    printf("%s", p->str);
}

//печать двоичного дерева
void printTree(TreeNode* root, Trunk *prev, int isLeft)
{
    if (root == NULL) {
        return;
    }

    char* prev_str = " ";
    Trunk* trunk = (Trunk*)malloc(sizeof(Trunk));
    trunk->prev = prev;
    trunk->str = prev_str;

    printTree(root->right, trunk, 1);

    if (!prev) {
        trunk->str = "~~~";
    }
    else if (isLeft)
    {
        char s1[20] = ".~~~";
        char s2[20] = ".~~~ ";
        trunk->str = s1;
        if (root->left == NULL) trunk->str = strcat(s2, root->data);
        prev_str = " |";
    }
    else {
        char s1[20] = "`~~~";
        char s2[20] = "`~~~ ";
        trunk->str = s1;
        if (root->right == NULL) trunk->str = strcat(s2, root->data);
        prev->str = prev_str;
    }

    showTrunks(trunk);
    printf("%c\n", root->oper);

    if (prev)
    {
        prev->str = prev_str;
    }
}

```

```

    }
    trunk->str = " |";

    printTree(root->left, trunk, 0);
}

int main(int argc, char* argv[])
{
    Stack.MaxSize = MaxStack;
    Stack.Size = 0;
    Stack.Head = NULL;
    char *formula = (char *)malloc(sizeof(char)*MaxString);
    char *res;

    printf("Enter your formula: ");
    gets(formula);
    printf("\nFormula:\n");
    printf(formula);
    printf("\n\n");
    //res = Postfix(buf, strlen(buf) + MaxStack);
    res = Postfix(formula, strlen(formula) + MaxStack);
    printf("Postfix record:\n");
    printf(res);
    printf("\n\n");

    printf("Solution tree:\n");
    TreeNode* SolutionTree = MakeSolutionTree(res);
    printTree(SolutionTree, NULL, 0);
    printf("\n\n");

    printf("Multiply reduced tree:\n");
    MultiplyReduceTree(SolutionTree);
    printTree(SolutionTree, NULL, 0);
    printf("\n");
    freemem(SolutionTree);

    return 0;
}

```

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

#листинг программы работы с деревом решений

```

// Реализовать алгоритм перевода из инфиксной записи арифметического выражения в
постфиксную
#include <stdio.h>
#include <malloc.h>
#include <ctype.h>
#include <string.h>
#define T char
#define MaxStack 36735895
#define MaxString 100

//операнды и скобки выражений
char OpenMatrix[3] = {'(', '[', '{'};
char CloseMatrix[3] = {')', ']', '}'};
char OperationMatrix[4] = {'+', '-', '*', '/'};

//формулы для тестирования программы
//char buf[MaxString] = "(140/2)-(30*(10-4))+(5*2)";
//char buf[MaxString] = "a * (5 - e) + g7 * 1 + ( b - c ) * d3";
//char buf[MaxString] = "(A+B)*(C+D)-E";
//char buf[MaxString] = "(a + 5) / 3*d";
char buf[MaxString] = "(a + 5) / (3 * d) - (c + (7 - e) * b) + 7 + 5*e";

//структура двоичного дерева
typedef struct tnode {      // узел дерева
    char oper;              // операнд (+ - * /)
    char* data;             // переменная или константа
    int count;              // число вхождений
    struct tnode* left;     // левый потомок
    struct tnode* right;    // правый потомок
    struct tnode* parent;   // родительский узел
} TreeNode;

typedef struct TNode
{
    T value;
    struct TNode *next;
} Node;

//стек операций для формирования постфиксной записи
struct TStack
{
    Node *Head;
    int Size;
    int MaxSize;
};

```

```
struct TStack Stack;
```

```
//структура для вывода дерева в консоль
```

```
typedef struct TTrunk
```

```
{  
    struct TTrunk* prev;  
    char* str;  
} Trunk;
```

```
//запись в стек
```

```
void Push(T item)
```

```
{  
    if (Stack.Size > MaxStack)  
    {  
        printf("Stack overflow!");  
    }  
    else  
    {  
        Node *tmp = (Node *)malloc(sizeof(Node));  
        if (tmp)  
        {  
            tmp->value = item;  
            tmp->next = Stack.Head;  
            Stack.Head = tmp;  
            Stack.Size++;  
        }  
        else  
        {  
            printf("Stack overflowed!");  
            Stack.Size = MaxStack + 1;  
        }  
    }  
}
```

```
//запрос и удаление из стека
```

```
T Pop()
```

```
{  
    if (Stack.Size == 0)  
    {  
        printf("Stack is empty!");  
        return NULL;  
    }  
    else  
    {  
        Node *node = Stack.Head;
```



```

        T item = Stack.Head->value;
        Stack.Head = Stack.Head->next;
        Stack.Size--;
        free(node);
        return item;
    }
}

```

```

void PrintStack(Node *node)
{
    while (node != NULL)
    {
        printf("%d", node->value);
        node = node->next;
    }
}

```

```

//проверка на открывающую скобку
int isOpenSymbol(char c)
{
    int i;
    for (i = 0; i < 3; i++)
    {
        if (c == OpenMatrix[i])
            return 1;
    }
    return 0;
}

```

```

//проверка на закрывающую скобку
int isCloseSymbol(char c)
{
    int i;
    for (i = 0; i < 3; i++)
    {
        if (c == CloseMatrix[i])
            return 1;
    }
    return 0;
}

```

```

//проверка на операнд + - * /
int isOperationSymbol(char c)
{
    int i;

```

```

for (i = 0; i < 4; i++)
{
    if (c == OperationMatrix[i])
        return 1;
}
return 0;
}

```

```

//проверка на число
int isNumber(char *str)
{
    if (str == NULL || strlen(str) == 0)
        return 0;
    int i = 0;
    while (str[i] != '\0')
        if (!isdigit(str[i++]))
            return 0;
    return 1;
}

```

```

//проверка на переменную
int isVariable(char *str)
{
    if (str == NULL || strlen(str) == 0 || !isalpha(str[0]))
        return 0;
    int i = 0;
    while (str[i] != '\0')
    {
        if (!isalpha(str[i]) && !isdigit(str[i]))
            return 0;
        i++;
    }
    return 1;
}

```

```

//приоритет операции
int PRIOR(char a)
{
    switch(a)
    {
        case '*':
        case '/':
            return 3;

        case '-':

```

```

    case '+':
        return 2;

    case '(':
        return 1;
}
}

// перевод из инфиксной записи арифметического выражения в постфиксную
char *Postfix(char *str, int str_len)
{
    char *PostfixString = (char *)malloc(sizeof(char) * str_len);
    strcpy(PostfixString, "\0");
    int i;
    for (i = 0; i < strlen(str); i++)
    {
        char el[20] = "";
        int j = 0;
        while (str[i] != ' ' && str[i] != '\0')
        {
            char tmp = str[i];

            if (isOpenSymbol(tmp))
            {
                Push(tmp);
                break;
            }
            else if (isOperationSymbol(tmp))
            {
                while (Stack.Size > 0 && PRIOR(Stack.Head->value) >= PRIOR(tmp))
                {
                    char oper = Pop();
                    char tmp_str[2];
                    tmp_str[0] = oper;
                    tmp_str[1] = 0;
                    strcat(PostfixString, tmp_str);
                    strcat(PostfixString, " ");
                }
                Push(tmp);
                break;
            }
            else if (isCloseSymbol(tmp))
            {
                if (isNumber(el) || isVariable(el))
                {

```

```

        strcat(PostfixString, el);
        strcat(PostfixString, " ");
        el[0] = '\0';
    }
    char oper;
    do
    {
        oper = Pop();
        if (oper != NULL && !isOpenSymbol(oper))
        {
            char tmp_str[2];
            tmp_str[0] = oper;
            tmp_str[1] = 0;
            strcat(PostfixString, tmp_str);
            strcat(PostfixString, " ");
        }
    } while (!isOpenSymbol(oper));

    if (!isOpenSymbol(oper))
    {
        return NULL;
    }

    j--;
}
else
{
    el[j] = tmp;
}

j++;
i++;
}
el[j] = '\0';
if (isNumber(el) || isVariable(el))
{
    strcat(PostfixString, el);
    strcat(PostfixString, " ");
}
}
while (Stack.Size > 0)
{
    char oper = Pop();
    char tmp_str[2];
    tmp_str[0] = oper;

```

```

    tmp_str[1] = 0;
    strcat(PostfixString, tmp_str);
    strcat(PostfixString, " ");
}
return PostfixString;
}

```

//разделение постфиксной записи на 2 части для формирования двоичного дерева

```

int SplitPostfix(char *str)
{
    int indx = strlen(str);
    int form = 1;
    int symb;
    int oper;
    char el;
    while (indx > 0 && form > 0)
    {
        el = str[indx-1];
        while (el == ' ')
        {
            indx--;
            el = str[indx-1];
        }

        symb = 0;
        oper = 0;
        while (indx > 0 && (isalpha(el) || isdigit(el)))
        {
            symb++;
            indx--;
            el = str[indx-1];
        }
        while (indx > 0 && isOperationSymbol(el))
        {
            oper++;
            indx--;
            el = str[indx-1];
        }

        if (symb > 0)
            form--;

        if (oper > 0)
            form++;
    }
}

```

```

        if (indx > 0 && form > 0) indx--;
    }
    return indx;
}

```

//удаление лишних пробелов

```

char* Trim(char* str)
{
    int indx = strlen(str);
    char* s = (char*)malloc(sizeof(char) * indx);
    strcpy(s, str);
    while (s[indx-1] == ' ')
    {
        s[indx-1] = '\0';
        indx--;
    }
    return s;
}

```

//формирования дерева решений из постфиксной записи арифметической формулы

```

TreeNode* MakeSolutionTree(char* formula)
{
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    node->parent = NULL;
    node->left = NULL;
    node->right = NULL;
    node->data = formula;
    node->count = 0;
    node->oper = ' ';

```

```

    int len = strlen(formula);
    char oper;
    char* f = (char *)malloc(sizeof(char) * len);
    char* s1 = (char *)malloc(sizeof(char) * len);
    char* s2 = (char *)malloc(sizeof(char) * len);

```

```

    strcpy(f, formula);

```

```

    while (f[len-1] != ' ') len--;
    oper = f[len-1];
    f[len-1] = 0;

```

```

    int split_indx = SplitPostfix(f);
    if (split_indx > 0 && isOperationSymbol(oper))
    {

```

```

        strncpy(s1, f, split_indx);
s1[split_indx-1] = '\0';
s1 = Trim(s1);

strcpy(s2, f+split_indx);
s2 = Trim(s2);

int k1 = 0;
    int k2 = 0;
node->oper = oper;
node->left = MakeSolutionTree(s1);
if (node->left != NULL)
{
    k1 = node->left->count+1;
    node->left->parent = node;
}
node->right = MakeSolutionTree(s2);
if (node->right != NULL)
{
    k2 = node->right->count+1;
    node->right->parent = node;
}
node->count = k1 > k2 ? k1 : k2;
}

return node;
}

```

// Функция удаления поддерева

```

void freemem(TreeNode* tree) {
    if (tree != NULL) {
        freemem(tree->left);
        freemem(tree->right);
        free(tree);
    }
}

```

//редуцирование операции умножения на сумму операций сложения

```

void MultiplyReduceTree(TreeNode* wood)
{
    if (wood != NULL)
    {
        if (wood->count == 1 && wood->oper == '*')
        {

```

```

        if ((isNumber(wood->right->data) && isVariable(wood->left->data)) ||
(isVariable(wood->right->data) && isNumber(wood->left->data)))
        {
            int num = isNumber(wood->right->data) ? atoi(wood->right-
>data) : atoi(wood->left->data);
            if (num == 0)
            {
                wood->data = "0\0";
                wood->oper = ' ';
                wood->count = 0;
                free(wood->left);
                free(wood->right);
                wood->left = NULL;
                wood->right = NULL;
            }
            else if (num == 1)
            {
                wood->oper = ' ';
                if (isNumber(wood->right->data))
                    wood->data = wood->left->data;
                else
                    wood->data = wood->right->data;
                wood->count = 0;
                free(wood->left);
                free(wood->right);
                wood->left = NULL;
                wood->right = NULL;
            }
            else if (num > 1)
            {
                char* str = (char *)malloc(sizeof(char) * 20);
                sprintf(str, "%d", num-1);

                TreeNode* oper_node =
(TreeNode*)malloc(sizeof(TreeNode));
                if (num < 4)
                    oper_node->oper = '+';
                else
                    oper_node->oper = wood->oper;
                oper_node->count = 1;
                oper_node->parent = wood;

                TreeNode* node1 =
(TreeNode*)malloc(sizeof(TreeNode));
                node1->oper = ' ';

```



```

        node1->count = 0;
        node1->parent = oper_node;

        TreeNode* node2 =
(TreeNode*)malloc(sizeof(TreeNode));
        node2->data = Trim(str);
        node2->oper = ' ';
        node2->count = 0;
        node2->parent = oper_node;

        if (isNumber(wood->right->data))
        {
            node1->data = wood->left->data;
            if (num < 4) node2->data = wood->left->data;
            wood->right = oper_node;
            oper_node->right = node2;
            oper_node->left = node1;
        }
        else
        {
            node1->data = wood->right->data;
            if (num < 4) node2->data = wood->right->data;
            wood->left = oper_node;
            oper_node->right = node1;
            oper_node->left = node2;
        }
        wood->oper = '+';
        wood->count = num-1;
    }
}

}
MultiplyReduceTree(wood->left);
MultiplyReduceTree(wood->right);
}
return;
}

```

//вспомогательная функция печати веток двоичного дерева

```
void showTrunks(Trunk *p)
```

```
{
    if (p == NULL) {
        return;
    }

```

```
    showTrunks(p->prev);

```

```

    printf("%s", p->str);
}

//печать двоичного дерева
void printTree(TreeNode* root, Trunk *prev, int isLeft)
{
    if (root == NULL) {
        return;
    }

    char* prev_str = " ";
    Trunk* trunk = (Trunk*)malloc(sizeof(Trunk));
    trunk->prev = prev;
    trunk->str = prev_str;

    printTree(root->right, trunk, 1);

    if (!prev) {
        trunk->str = "~~~";
    }
    else if (isLeft)
    {
        char s1[20] = ".~~~";
        char s2[20] = ".~~~ ";
        trunk->str = s1;
        if (root->left == NULL) trunk->str = strcat(s2, root->data);
        prev_str = " |";
    }
    else {
        char s1[20] = "`~~~";
        char s2[20] = "`~~~ ";
        trunk->str = s1;
        if (root->right == NULL) trunk->str = strcat(s2, root->data);
        prev->str = prev_str;
    }

    showTrunks(trunk);
    printf("%c\n", root->oper);

    if (prev)
    {
        prev->str = prev_str;
    }
    trunk->str = " |";
}

```

```

    printTree(root->left, trunk, 0);
}

int main(int argc, char* argv[])
{
    Stack.MaxSize = MaxStack;
    Stack.Size = 0;
    Stack.Head = NULL;
    char *formula = (char *)malloc(sizeof(char)*MaxString);
    char *res;

    printf("Enter your formula: ");
    gets(formula);
    printf("\nFormula:\n");
    printf(formula);
    printf("\n\n");
    //res = Postfix(buf, strlen(buf) + MaxStack);
    res = Postfix(formula, strlen(formula) + MaxStack);
    printf("Postfix record:\n");
    printf(res);
    printf("\n\n");

    printf("Solution tree:\n");
    TreeNode* SolutionTree = MakeSolutionTree(res);
    printTree(SolutionTree, NULL, 0);
    printf("\n\n");

    printf("Multiply reduced tree:\n");
    MultiplyReduceTree(SolutionTree);
    printTree(SolutionTree, NULL, 0);
    printf("\n");
    freemem(SolutionTree);

    return 0;
}

```

#компиляция программы

irina@Irina-VivoBook:~/Prog/Prog\_C/Lab24\$ gcc SolutionTree.c

#запуск программы с формулой -  $(b + 3) / (c * 2)$

Enter your formula:  $(b + 3) / (c * 2)$

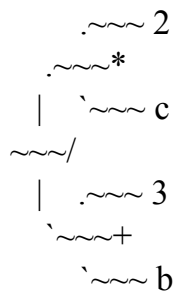
Formula:

$(b + 3) / (c * 2)$

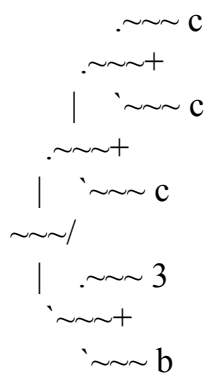
Postfix record:

b 3 + c 2 \* /

Solution tree:



Multiply reduced tree:



#запуск программы с формулой (b + 3) / c \* 2

irina@Irina-VivoBook:~/Prog/Prog\_C/Lab24\$ ./a.out

Enter your formula: (b + 3) / c \* 2

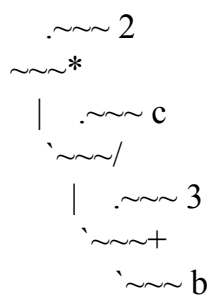
Formula:

(b + 3) / c \* 2

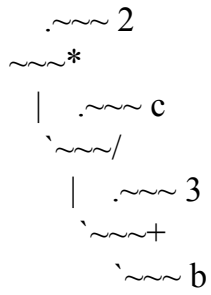
Postfix record:

b 3 + c / 2 \*

Solution tree:



Multiply reduced tree:



#запуск программы с формулой  $3 + 6*c$

irina@Irina-VivoBook:~/Prog/Prog\_C/Lab24\$ ./a.out

Enter your formula:  $3 + 6*c$

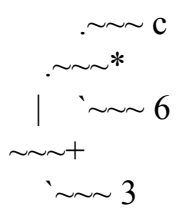
Formula:

$3 + 6*c$

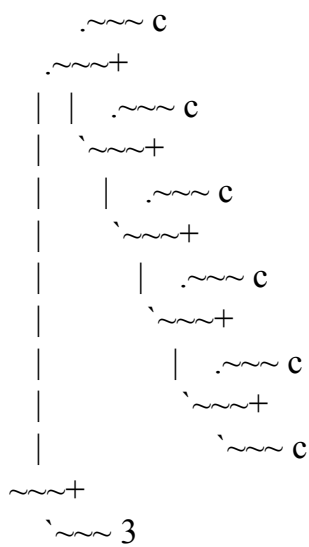
Postfix record:

3 6 c \* +

Solution tree:



Multiply reduced tree:



9. 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.

10. Замечания автора по существу работы: замечания отсутствуют.

11. Выводы: научился работать с двоичными деревьями и обрабатывать выражения заданным образом.

Подпись студента \_\_\_\_\_