



## Отчет по лабораторной работе № 24 по курсу Алгоритмы и структуры данных

Студент группы М8О-103Б-22 Ахметшин Булат Рамилевич, № по списку 2

Контакты www, e-mail, icq, skype ahmbulat04@yandex.ru

Работа выполнена: 31.05.2023 г.

Преподаватель: доцент каф. 806 Никулин С.П.

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

Отчет сдан «    » \_\_\_\_\_ 202 \_\_ г., итоговая оценка \_\_\_\_

Подпись преподавателя \_\_\_\_\_

1. **Тема:** Деревья выражений. \_\_\_\_\_

2. **Цель работы:** Научиться реализовывать алгоритмы построения и обработки деревьев выражений. \_\_\_\_\_

3. **Задание (вариант № 3):** Составить программу выполнения заданных преобразований арифметических выражений с применением деревьев на языке Си. \_\_\_\_\_

4. **Оборудование (лабораторное):**

ЭВМ \_\_\_\_\_, процессор \_\_\_\_\_, имя узла сети \_\_\_\_\_ с ОП \_\_\_\_\_ Мб,  
НМД \_\_\_\_\_ Мб. Терминал \_\_\_\_\_ адрес \_\_\_\_\_. Принтер \_\_\_\_\_  
Другие устройства \_\_\_\_\_

*Оборудование ПЭВМ студента, если использовалось:*

Процессор Intel(R) Core(TM) i7-10510U с ОП 8 ГБ НМД SSD 512 ГБ . Монитор Встроенный 1920x1080  
Другие устройства \_\_\_\_\_

5. **Программное обеспечение (лабораторное):**

Операционная система семейства \_\_\_\_\_, наименование \_\_\_\_\_ версия \_\_\_\_\_  
интерпретатор команд \_\_\_\_\_ версия \_\_\_\_\_  
Система программирования \_\_\_\_\_ версия \_\_\_\_\_  
Редактор текстов \_\_\_\_\_ версия \_\_\_\_\_  
Утилиты операционной системы \_\_\_\_\_

Прикладные системы и программы \_\_\_\_\_

Местонахождение и имена файлов программ и данных \_\_\_\_\_

*Программное обеспечение ЭВМ студента, если использовалось:*

Операционная система семейства UNIX, наименование Ubuntu версия 22.04  
интерпретатор команд GNU bash версия 5.1.16  
Система программирования Visual Studio Code версия 1.77.3  
Редактор текстов Sublime Text 3 версия 3211  
Утилиты операционной системы Стандартные утилиты OS Linux

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

Программа будет рассматривать выражения, которые представляют из себя суперпозицию 7 операций, 2 из которых унарные (унарные  $+$ ,  $-$ ) и 5 бинарных ( $+$ ,  $-$ ,  $*$ ,  $/$ ,  $^$ ), операндами которых могут быть вещественные числа с разделителем '.' (например, 5.47), переменные, которые представляют символы латинского алфавита ( $a, b, \dots, x, y, z$ ), и скобочные подвыражения.

Каждой операции будет соответствовать приоритет в виде целого числа, начиная с единицы; также каждой операции в выражении будет соответствовать приоритет глубины (priority depth - pd), который равен десяти суммам открытых скобок до самой операции (т.е. приоритет глубины выражает собой глубину скобочного подвыражения, таким образом, приоритет глубины  $*$  в выражении  $a + (b + (c * d))$  будет равен  $10 \cdot (1 + 1) = 20$ ). Абсолютным приоритетом будем называть сумму операции и её приоритет глубины.

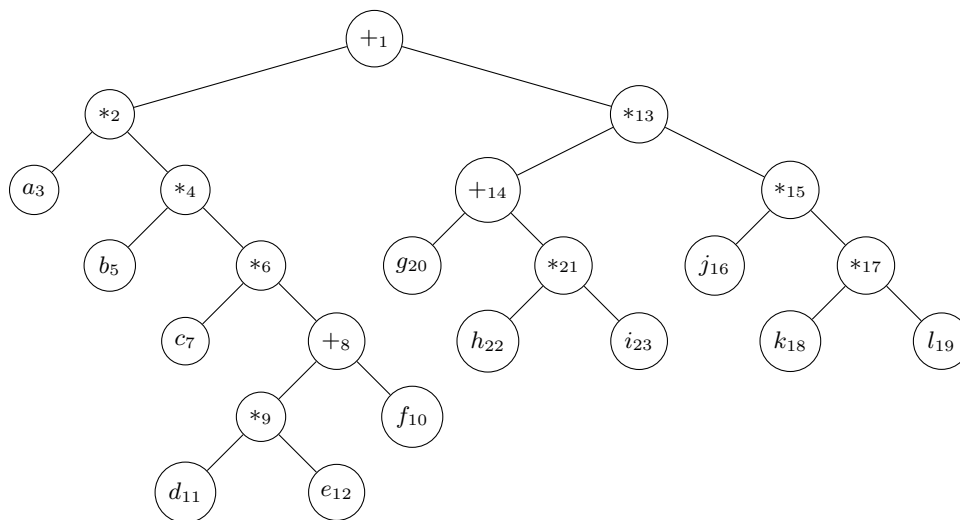
Дерево выражений будет строиться следующим образом:

- Каждой операции соответствует вершина дерева выражений
- Левое поддерево вершины операции есть левый операнд
- Правое поддерево вершины операции есть правый операнд
- Если выражение не содержит операций, то единственная вершина дерева представляет число или переменную
- Чем больше абсолютный приоритет операции - тем больше глубина её вершины
- Преобразование выражения в дерево происходит слева-направо

Из этих правил следует, что в дереве выражений листьями могут быть только числа и переменные, остальными вершинами дерева будут операции, притом подвыражения суперпозиции операций одного абсолютного приоритета выстраиваются в 'линию'.

Рассмотрим пример:

Выражение  $a * b * c * (d * e + f) + (g + h * i) * j * k * l$  будет преобразовано в дерево:



(Каждой вершине соответствует номер в индексе символа этой вершины)

Сначала будет добавлена вершина 3, затем будет добавлен её предок 2, после в правый лист 2 добавится вершина 5; следующая операция 4, чей абсолютный приоритет равен абсолютному приоритету предыдущей операции, будет помещена в правое поддерево вершины 2, а вершина 5 поместится в левое поддерево вершины 4.

После того, как в правое поддерево вершины 6 добавиться вершина 11, будет считана операция 9. Её абсолютный приоритет выше, чем операции 6, она замещает вершину 11, а вершина 11 добавляется в левое поддерево 9 вершины, после в правое поддерево той же вершины будет добавлена вершина 12. Следующая операция 8 - её приоритет ниже операции 9, поэтому она помещается в правое поддерево вершины 6, а в левое поддерево вершины 8 перемещается вершина 9, затем в левое поддерево вершины 8 добавляется вершина 10.

Итак, поддерево с корнем в вершине 2 построено. После этого будет считана операция 1, её абсолютный приоритет ниже, чем любой другой операции в построенном поддереве, следовательно, она станет предком вершины 2, которая, в свою очередь, станет левым поддеревом вершины 1.

Левое поддерево вершины 1 строится аналогично.

Печать дерева реализуется следующим алгоритмом:

- (a) Печать левого поддерева
- (b) Печать вершины
- (c) Печать правого поддерева

Печать скобок будет осуществляться за счет определения приоритета глубины текущей операции. Если она больше приоритета глубины предыдущей операции, справа и слева от крайних членов скобочного выражения должны быть напечатаны открывающая и закрывающая скобки соответственно.

Перемножение чисел в членах выражения будет производиться в два этапа:

- (a) Перенос констант в начало члена
- (b) Последовательное перемножение констант в каждом члене

Для переноса констант нужно рекурсивно искать 'линии' (последовательности операций одного приоритета глубины с их операндами) в выражении, и производить перенос в этих линиях. Т.к. при выполнении рекурсивной функции, относительное положение вершины, на адрес которой указывает данный указатель, меняется, необходимо передавать начало и конец перестроенной 'линии' и продолжать выполнение алгоритма с учетом этих данных.

После переноса констант в членах выражения, нужно их перемножить. Для этого нужно снова отыскать 'линии' и заменить подвыражения произведения констант на одну константу (результат произведения).

Второй алгоритм при произвольной операции представляет собой упрощение выражения для произвольной (в том числе некоммутативной) операции.

Преобразование, соответствующее моему варианту, я решил реализовать как композицию двух описанных выше алгоритмов.

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

- (a) Написать функцию проверки валидности выражения
- (b) Написать функцию парсинга валидного выражения и построения соответствующего дерева
- (c) Написать функцию печати дерева выражения
- (d) Написать функцию переноса констант в 'линиях' выражения вперед
- (e) Написать функцию упрощения выражения для произвольной операции
- (f) Написать функцию упрощения коммутативной операции (композиция двух предыдущих функций)

*Пункты 1-7 отчета составляются строго до начала лабораторной работы.*

*Допущен к выполнению работы. Подпись преподавателя \_\_\_\_\_*

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

```
bulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$ ls
expr_tree.c  expr_tree.h  124-2012.djvu  logs  main.c
bulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$ cat expr_tree.h

#ifndef __EXPR_TREE_H
#define __EXPR_TREE_H

#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

// Operations

typedef enum {
    // unary minus and plus
    _EXPR_UN_MINUS,
    _EXPR_UN_PLUS,
    // binary minus and plus
    _EXPR_BIN_MINUS,
    _EXPR_BIN_PLUS,
    // binary multiplication and division
    _EXPR_BIN_MULT,
    _EXPR_BIN_DIV,
    // binary power
    _EXPR_BIN_POW,
    // case if node is not an operation
    _EXPR_NOP
} _EXPR_OPERATION;

// Type of expression tree node
typedef enum {
    _EXPR_VAR,
    _EXPR_CONST,
    _EXPR_OP,
    _NONTYPE
} _EXPR_EL_TYPE;

typedef union {
    double num;
    char var;
    char op;
} expr_node_data;

// Node of expression tree
typedef struct expr_node {
```

```

    _EXPR_EL_TYPE type;
    _EXPR_OPERATION op;
    expr_node_data data;
    struct expr_node* left;
    struct expr_node* right;
    struct expr_node* ancestor;
    // Priority depth of the operation (for brackets)
    // if node is an operation
    uint8_t pd;
} expr_node;

typedef struct expr_line {
    expr_node* p1;
    expr_node* p2;
} expr_line;

// Expression tree
typedef struct expr_tree {
    // count of nodes in tree
    int64_t n;
    // tree root
    expr_node* root;
} expr_tree;

int8_t is_valid_expression(char* const e, uint32_t _size);

// Return expr_node_data variable with memory filled with 0
expr_node_data expr_node_data_dv();

void construct_empty_expr_tree(expr_tree* t);

void construct_empty_expr_node(expr_node* p);

void init_empty_expr_node(expr_node** p,
    _EXPR_EL_TYPE type,
    _EXPR_OPERATION op,
    expr_node* const ancestor);

void parse_expr(char* const expr, uint32_t _size, expr_tree* t);

void free_expr_tree(expr_node* root);

void print_expr_tree(expr_node* root, int8_t bl, int8_t br);

void simplify_bin_operation(expr_node* root, _EXPR_OPERATION op);

void const_fwrd_by_op_in_line(expr_node** p1, expr_node** p2, _EXPR_OPERATION op);

expr_node* const_fwrd_by_operation(expr_node* root, _EXPR_OPERATION op);

expr_node* simplify_com_operation(expr_node* root, _EXPR_OPERATION op);

#endifbulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$ cat expr_tree.c

#include "expr_tree.h"

int8_t is_operation(char c) {
    switch (c)
    {
        case '-':
            return 1;
        case '+':
            return 1;
        case '*':
            return 1;
        case '/':
            return 1;
        case '^':
            return 1;
        default:
            return 0;
    }
}

int8_t is_commutative(_EXPR_OPERATION op) {
    switch (op)
    {
        case _EXPR_BIN_DIV:
            return 1;
        case _EXPR_BIN_MULT:
            return 1;
        case _EXPR_BIN_MINUS:
            return 1;
        case _EXPR_BIN_PLUS:
            return 1;
    }
}

```

```

        default:
            return 0;
    }
}

int8_t expr_op_is_unary(_EXPR_OPERATION op) {
    return op == _EXPR_UN_PLUS || op == _EXPR_UN_MINUS;
}

_EXPR_OPERATION operation(char c, int8_t unary) {
    if (c == '-' && unary) {
        return _EXPR_UN_MINUS;
    } else if (c == '+' && unary) {
        return _EXPR_UN_PLUS;
    } else if (c == '-') {
        return _EXPR_BIN_MINUS;
    } else if (c == '+') {
        return _EXPR_BIN_PLUS;
    } else if (c == '*') {
        return _EXPR_BIN_MULT;
    } else if (c == '/') {
        return _EXPR_BIN_DIV;
    } else if (c == '^') {
        return _EXPR_BIN_POW;
    } else {
        return _EXPR_NOP;
    }
}

// priority of operations
int8_t expr_op_prty(_EXPR_OPERATION op) {
    switch (op)
    {
        case _EXPR_BIN_MINUS:
            return 1;
        case _EXPR_BIN_PLUS:
            return 1;
        case _EXPR_BIN_MULT:
            return 2;
        case _EXPR_BIN_DIV:
            return 2;
        case _EXPR_UN_MINUS:
            return 3;
        case _EXPR_UN_PLUS:
            return 3;
        case _EXPR_BIN_POW:
            return 4;
        case _EXPR_NOP:
            return 0;
        default:
            return -1;
    }
}

int8_t is_valid_expression(char* const _e, uint32_t _size) {
    char* p = _e;
    if (p == NULL) {
        return 0;
    }

    char* _p = (char*)malloc(sizeof(char) * _size);
    p = _p;
    uint32_t size = 0;
    for (uint32_t i = 0; i < _size; ++i) {
        if (_e[i] != ' ') {
            p[size++] = _e[i];
        }
    }

    int8_t res = 1;

    // Is first letter, count of parentheses not closed
    int8_t fl = 1, psc = 0;
    while (*p != '\0') {
        // There should be no other operation and after could be only digit, parenthesis or variable
        if (is_operation(*p)) {
            // Incorrect what's after
            if (*(p + 1) == '\0' ||
                !(isdigit(*(p + 1)) || *(p + 1) == '(' || isalpha(*(p + 1)))) {
                res = 0; break;
            }
            // Incorrect what's before if
            // unary
            if (fl && !(*p == '-' || *p == '+')) {
                res = 0; break;
            }
        }
    }
}

```

```

    }
    // binary
    if (!( *p == '-' || *p == '+' ) && (*p - 1) == '(' || is_operation(*p - 1)) {
        res = 0; break;
    }
    // Variable or first and has operation after or last and has operation before
    // or has operation before and after or single
    } else if (isalpha(*p)) {
        if (!(f1 || is_operation(*p - 1) || *p - 1 == '(')) {
            res = 0; break;
        }
        if (!(f1 || ((*p + 1) == '\0') || is_operation(*p + 1) || *p + 1 == ')')) {
            res = 0; break;
        }
    }
    // Number is just as variable but it could be longer than one symbol
    } else if (isdigit(*p)) {
        if (!(f1 || is_operation(*p - 1) || *p - 1 == '(')) {
            res = 0; break;
        }
        int8_t count_of_dots = 0;
        while (*p != '\0' && (isdigit(*p) || *p == '.')) {
            if (*p == '.') {
                ++count_of_dots;
                if (1 < count_of_dots) {
                    res = 0; break;
                }
            }
            ++p;
        }
        if (!res) {
            break;
        }
        --p;
        if (!( (*p + 1) == '\0' || is_operation(*p + 1) || *p + 1 == ')')) {
            res = 0; break;
        }
    }
    // Open bracket should has operation before it if not first symbol
    // and should not be last and should not has another open bracket after it
    } else if (*p == '(') {
        if (!(f1 || is_operation(*p - 1) || *p - 1 == '(')) {
            res = 0; break;
        }
        if ((*p + 1) == '\0' || *p + 1 == ')') {
            res = 0; break;
        }
        ++p;
    }
    // Close bracket should not be first, has operation or open bracker before it and should be last
    // or has operation after it
    } else if (*p == ')') {
        if (f1 || is_operation(*p - 1) || *p - 1 == '(') {
            res = 0; break;
        }
        if (!( *p + 1 == '\0' || is_operation(*p + 1) || *p + 1 == ')')) {
            res = 0; break;
        }
        --p;
    } else {
        res = 0; break;
    }
    f1 = 0;
    ++p;
}

free(_p);

return res && psc == 0;
}

expr_node_data expr_node_data_dv() {
    expr_node_data _;
    _ . num = 0;
    return _;
}

void construct_empty_expr_tree(expr_tree* t) {
    t->n = 0;
    t->root = NULL;
}

void construct_empty_expr_node(expr_node* p) {
    init_empty_expr_node(&p, _NONTYPE, _EXPR_NOP, NULL);
}

void init_empty_expr_node(expr_node** p,
    _EXPR_EL_TYPE type,

```

```

        _EXPR_OPERATION op,
        expr_node* const ancestor) {
    *p = (expr_node*)malloc(sizeof(expr_node));
    (*p)->type = type;
    (*p)->op = op;
    (*p)->ancestor = ancestor;
    (*p)->data.var = 0;
    (*p)->pd = 0;
    (*p)->left = NULL;
    (*p)->right = NULL;
}

double expr_make_num(char** _c) {
    char** c = _c;
    double f = 0, i = 10;
    int8_t phase = 1;
    while (**c != '\0' && (isdigit(**c) || **c == '.')) {
        if (**c == '.') {
            phase = 2;
            ++(*c);
        }
        if (phase == 1) {
            f = 10 * f + (**c - 48);
        } else if (phase == 2) {
            f += (double)(**c - 48) / i;
            i *= 10;
        }
        ++(*c);
    }
    --(*c);
    return f;
}

void parse_expr(char* const _expr, uint32_t _size, expr_tree* t) {
    char* expr = (char*)malloc(sizeof(char) * _size);
    uint32_t size = 0;
    for (uint32_t i = 0; i < _size; ++i) {
        if (_expr[i] != ' ') {
            expr[size++] = _expr[i];
        }
        if (_expr[i] == '\0') {
            break;
        }
    }

    construct_empty_expr_tree(t);

    expr_node* p = NULL;

    int8_t empty = 1;

    // Priority depth. Each time parenthesis opens priority depth
    // of next operations increases and decreases when closes
    int8_t pd = 0;

    // Priority of last operation
    int8_t lp = 0;

    char* c = expr, *pr = NULL;
    while (*c != '\0') {
        // Case if read symbol was a letter means that we're reading a variable
        if (isalpha(*c)) {
            // If tree is empty we should create root
            if (empty) {
                init_empty_expr_node(&p, _EXPR_VAR, _EXPR_NOP, NULL);
                p->data.var = *c;
                t->root = p;
                empty = 0;
            } else {
                // Variable or constant can be read before operation only in the beginning
                init_empty_expr_node(&p->right, _EXPR_VAR, _EXPR_NOP, p);
                p = p->right;
                p->data.var = *c;
            }
        }
        // Case if readen symbol is '-', '+', '*', '/' or '^' means that we're reading an operation
        } else if (is_operation(*c)) {
            // Unary
            if (p == NULL || p->type == _EXPR_OP) {
                if (p == NULL) {
                    init_empty_expr_node(&p, _EXPR_OP, operation(*c, 1), NULL);
                    p->data.op = *c;
                    p->pd = pd;
                    t->root = p;
                    empty = 0;
                }
            }
        }
        ++c;
    }
}

```



```

        lp = expr_op_prty(operation(*c, 1)) + pd;
    } else {
        init_empty_expr_node(&p->right, _EXPR_OP, operation(*c, 1), p);
        p = p->right;
        p->data.op = *c;
        p->pd = pd;
        lp = expr_op_prty(operation(*c, 1)) + pd;
    }
}
// Binary
} else {
    if (expr_op_prty(operation(*c, 0)) + pd < lp) {
        while (! (p->ancestor == NULL) &&
            expr_op_prty(operation(*c, 0)) + pd < expr_op_prty(p->ancestor->op) + p->ancestor->pd) {
            p = p->ancestor;
        }
        expr_node* _p = p->ancestor;
        init_empty_expr_node(&p->ancestor, _EXPR_OP, operation(*c, 0), NULL);
        p->ancestor->left = p;
        p = p->ancestor;
        if (! (_p == NULL)) {
            _p->right = p;
            p->ancestor = _p;
        }
        p->data.op = *c;
        p->pd = pd;
        if (p->ancestor == NULL) {
            t->root = p;
        }
        lp = expr_op_prty(operation(*c, 0)) + pd;
    } else {
        expr_node* _p;
        init_empty_expr_node(&_p, _EXPR_OP, operation(*c, 0), p->ancestor);
        _p->data.op = *c;
        _p->pd = pd;
        if (! (p->ancestor == NULL)) {
            p->ancestor->right = _p;
        }
        _p->left = p;
        p->ancestor = _p;
        p = _p;
        if (p->ancestor == NULL) {
            t->root = p;
        }
        lp = expr_op_prty(operation(*c, 0)) + pd;
    }
}
}
// Case if read sybmol was a bracket
} else if (*c == '(') {
    pd += 10;
} else if (*c == ')') {
    pd -= 10;
}
// Cse if read symbol was a digit
} else if (isdigit(*c)) {
    // If tree is empty we should create root
    if (empty) {
        init_empty_expr_node(&p, _EXPR_CONST, _EXPR_NOP, NULL);
        p->data.num = expr_make_num(&c);
        t->root = p;
        empty = 0;
    } else {
        // Variable or constant can be read before operation only in the beggining
        init_empty_expr_node(&p->right, _EXPR_CONST, _EXPR_NOP, p);
        p = p->right;
        p->data.num = expr_make_num(&c);
    }
}
}
}
}

void print_expr_tree(expr_node* root, int8_t bl, int8_t br) {
    if (root == NULL) {
        return;
    }
    if (root->type == _EXPR_OP) {
        if (! (root->ancestor == NULL) &&
            root->ancestor->pd < root->pd) {
            print_expr_tree(root->left, bl + 1, 0);
        } else {
            print_expr_tree(root->left, bl, 0);
        }
    }
    if (root->type == _EXPR_OP) {
        if (expr_op_is_unary(root->op)) {
            for (int8_t i = 0; i < bl + 1; ++i) {

```

```

        printf("(");
    }
    printf("%c", root->data.op);
} else {
    printf(" %c ", root->data.op);
}
} else if (root->type == _EXPR_VAR) {
    for (int8_t i = 0; i < bl; ++i) {
        printf("(");
    }
    printf("%c", root->data.var);
    for (int8_t i = 0; i < br; ++i) {
        printf(")");
    }
} else if (root->type == _EXPR_CONST) {
    for (int8_t i = 0; i < bl; ++i) {
        printf("(");
    }
    if ((root->data.num - (int)root->data.num) < 1e-2) {
        printf("%d", (int)root->data.num);
    } else {
        printf("%.2f", root->data.num);
    }
    for (int8_t i = 0; i < br; ++i) {
        printf(")");
    }
}
if (root->type == _EXPR_OP) {
    if ((! (root->ancestor == NULL) &&
        root->ancestor->pd < root->pd) ||
        expr_op_is_unary(root->op)) {
        print_expr_tree(root->right, 0, br + 1);
    } else {
        print_expr_tree(root->right, 0, br);
    }
}
}

void free_expr_tree(expr_node* root) {
    if (root == NULL) {
        return;
    }
    free_expr_tree(root->left);
    free_expr_tree(root->right);
    (*root).left = NULL;
    (*root).right = NULL;
    free(root);
}

void make_operation(expr_node* p1, expr_node *p2, _EXPR_OPERATION op) {
    switch (op)
    {
    case _EXPR_BIN_MULT:
        p1->data.num *= p2->data.num;
        break;
    case _EXPR_BIN_DIV:
        if (p1->data.num == 0) {
            printf("Division by zero error: %.4f / %.4f\n",
                p1->data.num, p2->data.num);
            exit(-1);
        }
        p1->data.num /= p2->data.num;
        break;
    case _EXPR_BIN_PLUS:
        p1->data.num += p2->data.num;
        break;
    case _EXPR_BIN_MINUS:
        p1->data.num -= p2->data.num;
        break;
    case _EXPR_NOP:
        break;
    default:
        printf("Wrong operation: %d\n", op);
        break;
    }
}

void simplify_simple_operation(expr_node* p1, expr_node *p2, _EXPR_OPERATION op) {
    if (! (p2 == NULL)) {
        while (1) {
            make_operation(p1->left, p1->right->left, op);
            expr_node* _p = p1->right;
            p1->right = p1->right->right;
            p1->right->ancestor = p1;
            if (_p == p2) {

```

```

        free(_p);
        break;
    } else {
        free(_p);
    }
}

if (p1->right->type == _EXPR_CONST) {
    p1->type = _EXPR_CONST;
    p1->op = _EXPR_NOP;
    p1->pd = 0;
    make_operation(p1->left, p1->right, op);
    p1->data.num = p1->left->data.num;
    free(p1->left);
    free(p1->right);
    p1->left = NULL;
    p1->right = NULL;
}
}

void _simplify_bin_operation(expr_node* root, _EXPR_OPERATION op, expr_node* p1, expr_node* p2) {
    if (root == NULL) {
        return;
    }
    // If node is an operation
    if (root->type == _EXPR_OP) {
        _simplify_bin_operation(root->left, op, NULL, NULL);
        // If is simple operation
        if (root->op == op && root->left->type == _EXPR_CONST) {
            if (p1 == NULL) {
                _simplify_bin_operation(root->right, op, root, NULL);
            } else {
                _simplify_bin_operation(root->right, op, p1, root);
            }
        } else if (! (p2 == NULL)) {
            simplify_simple_operation(p1, p2, op);
            _simplify_bin_operation(root->right, op, NULL, NULL);
        } else {
            _simplify_bin_operation(root->right, op, p1, NULL);
        }
    } else if (! (p1 == NULL)) {
        simplify_simple_operation(p1, p2, op);
    }
}

void simplify_bin_operation(expr_node* root, _EXPR_OPERATION op) {
    _simplify_bin_operation(root, op, NULL, NULL);
}

expr_line _const_fwrdb_by_operation(expr_node* root, _EXPR_OPERATION op, expr_node* p1, expr_node* p2) {
    if (root == NULL) {
        return (expr_line){ .p1 = NULL, .p2 = NULL };
    }
    if (root->type == _EXPR_OP) {
        if (root->op == op) {
            expr_line l;
            if (! (p1 == NULL)) {
                l = _const_fwrdb_by_operation(root->right, op, p1, root);
            } else {
                l = _const_fwrdb_by_operation(root->right, op, root, NULL);
            }
            p1 = l.p1, p2 = l.p2;
            if (! (p1 == NULL)) {
                if (p2 == NULL) {
                    _const_fwrdb_by_operation(p1->left, op, NULL, NULL);
                    _const_fwrdb_by_operation(p1->right, op, NULL, NULL);
                } else if (p2->right->type == _EXPR_OP) {
                    _const_fwrdb_by_operation(p2->right, op, NULL, NULL);
                }
            }
        } else {
            _const_fwrdb_by_operation(root->left, op, NULL, NULL);
            _const_fwrdb_by_operation(root->right, op, NULL, NULL);
        }
    } else if (! (p2 == NULL)) {
        const_fwrdb_by_op_in_line(&p1, &p2, op);
        return (expr_line){ .p1 = p1, .p2 = p2 };
    }
    return (expr_line){ .p1 = NULL, .p2 = NULL };
}

// Move constants forward in expression like a1 * a2 * a3 * ... * an, where some of a1,...,an are constants
// Should be used only for commutative operations
void const_fwrdb_by_op_in_line(expr_node** p1, expr_node** p2, _EXPR_OPERATION op) {

```

```

expr_node* p = *p1;

while (1) {
    _const_fwrd_by_operation(p->left, op, NULL, NULL);
    if (p == *p2) {
        break;
    }
    p = p->right;
}

expr_node* begin = *p1;
p = begin->right;

expr_node* to;
while (p->type == _EXPR_OP) {
    if (p == *p2) {
        *p2 = (*p2)->right;
    }
    if (p->left->type == _EXPR_CONST && p->pd == p->ancestor->pd) {
        p->right->ancestor = p->ancestor;
        p->ancestor->right = p->right;

        if (begin->left->type == _EXPR_CONST) {
            begin->right->ancestor = p;
            to = p->right;
            p->right = begin->right;
            begin->right = p;
            p->ancestor = begin;
        } else {
            p->ancestor = begin->ancestor;
            if (! (begin->ancestor == NULL)) {
                if (begin->ancestor->right == begin) {
                    begin->ancestor->right = p;
                } else {
                    begin->ancestor->left = p;
                }
            }
            begin->ancestor = p;
            to = p->right;
            p->right = begin;
            begin = p;
        }
    } else {
        to = p->right;
    }
    if (! ((*p2)->type == _EXPR_OP)) {
        *p2 = (*p2)->ancestor;
    }
    p = to;
}
p = p->ancestor;
if (p->right->type == _EXPR_CONST && p->pd == p->ancestor->pd) {
    *p2 = (*p2)->ancestor;
    p->ancestor->right = p->left;
    p->left->ancestor = p->ancestor;

    p->left = p->right;
    p->right = begin->left;
    begin->left->ancestor = p;

    begin->left = p;
    p->ancestor = begin;
}
*p1 = begin;
}

expr_node* back_to_root(expr_node* p) {
    if (! (p->ancestor == NULL)) {
        p = p->ancestor;
    } else {
        return p;
    }
    return back_to_root(p);
}

expr_node* const_fwrd_by_operation(expr_node* root, _EXPR_OPERATION op) {
    _const_fwrd_by_operation(root, op, NULL, NULL);
    return back_to_root(root);
}

// Simplify expression by commutative operation
expr_node* simplify_com_operation(expr_node* root, _EXPR_OPERATION op) {
    root = const_fwrd_by_operation(root, op);
    simplify_bin_operation(root, op);
}

```

```

    return root;
}bulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$ cat main.c

#include <stdio.h>
#include <stdlib.h>

#include "expr_tree.h"

int main() {

    int n;
    scanf("%d", &n);

    expr_tree t;

    for (int i = 0; i < n; ++i) {
        char* s = (char*)malloc(sizeof(char) * 50);
        scanf("%49[^\n]", s);
        if (is_valid_expression(s, 49)) {
            parse_expr(s, 49, &t);
            t.root = simplify_com_operation(t.root, _EXPR_BIN_MULT);
            print_expr_tree(t.root, 0, 0);
            printf("\n");
            free_expr_tree(t.root);
        } else {
            printf("Expression is not valid\n");
        }
        free(s);
    }

    return 0;
}bulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$ ls

expr_tree.c  expr_tree.h  124-2012.djvu  logs  main.c
bulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$ gcc -g main.c expr_tree.c
-o main

bulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$ ./main

10
2*b*2
4 * b
b*2*2
4 * b
2*3*b*4*5
120 * b
2*(3*4+a)*b*(5*6+c)*7*8
112 * (12 + a) * b * (30 + c)
1*2*x*3*4+5*6*y*7*8
24 * x + 1680 * y
-a*2*3*(9*10)
6 * (-a) * 90
-a*2*3*(9*10+b)
(-a) * 6 * (90 + b)
2*3*(-a)*(9*10+b)
6 * (-a) * (90 + b)
1 * 2 * 3 * 4 * 5 * x * 6 * 7 * 8 * 9
362880 * x
0 * a * 8 * 9
0 * a
bulat@bulat-Swift-SF314-58:~/Studying/prprm/1/124/ty2$

```

**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 ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы.

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

**10. Замечания автора** по существу работы: \_\_\_\_\_

-

-

**11. Выводы:** в ходе этой лабораторной работы я получил опыт реализации алгоритмов над деревьями выражений, в том числе их построение, вывод и преобразование.

-

-

**12. Недочёты** при выполнении задания могут быть устранены следующим образом: \_\_\_\_\_

-

Подпись студента \_\_\_\_\_