МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики

Кафедра теоретичної та прикладної інформатики

Індивідуальне завдання № 2

з курсу «Алгоритми і структури даних»

на тему: «Бінарні дерева»

Виконала: студентка 2 курсу групи МФ-22

напряму підготовки (спеціальності)

Комп’ютерні науки

122 Комп’ютерні науки

Мухачова В.В.

Харків – 2024

Завдання:

Формулу виду

<формула>::=<термінал>|(<формула><знак><формула>)

<знак>::= + | - | *

<термінал>::=<цифра>|<змінна>

<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

< змінна >::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

можна подати у вигляді двійкового дерева ("дерева-формули") згідно з наступними правилами: формула з однієї цифри представляється деревом з однієї вершини з цією цифрою, а формула виду (f1 s f2) – деревом, у якому корінь – це знак s, а ліве та праве піддерева – це відповідні подання формул f1 та f2.

a) Спростити дерево-формулу T, замінюючи в ньому всі дерева, що відповідають формулам (f+0), (0+f), (f-0), (f*1), (1*f), на піддерева, які відповідають формулі f, а піддерева, які відповідають формулам (f*0) і (0*f), – на вершину з 0.

b) Перетворити дерево-формулу T, виконуючи заміну в ньому: піддерева, які відповідають формулі на піддерева, які відповідають формулі

((f1+f2)*f3) —> ((f1*f3)+(f2*f3))

((f1-f2)*f3) —>((f1*f3)-(f2*f3))

(f1*(f2+f3)) —>((f1*f2)+(f1*f3))

(f1*(f2-f3)) —>((f1*f2)-(f1*f3))

Аналіз коду:

```
typedef enum
{
    Sign, Terminal
}NodeType;
```

Структура дає вибір між знаком та терміналом, для спрощення структури дерева.

```
typedef struct Node
{
    NodeType type;
    char symbol;
    struct Node* left;
    struct Node* right;
} Node;
```

Стуктура Node описує елемент дерева: тип елемента (знак або терминал), символ та вказівники на ліве та праве піддерево.

```
//спрощення дерева за правилами:(f+0), (0+f), (f-0), (f*1), (1*f), (f*0), (0*f)
Node* simplify(Node* root)
{
    if (root == NULL)
        return NULL;

    //спрощуємо праве та ліве піддерево
    root->left = simplify(root->left);
    root->right = simplify(root->right);

    //якщо це знак, то перевіряємо знак та піддерева
    if (root->type == Sign)
    {
        if (root->symbol == '+')
        {
            if (root->left!=NULL && root->left->type == Terminal && root->left->symbol == '0')
            {
                //(0+f)->f
                return root->right;
            }
            if (root->right!=NULL && root->right->type == Terminal && root->right->symbol == '0')
            {
                //(f+0)->f
                return root->left;
            }
        }
        else if (root->symbol == '-')
        {
            if (root->right!=NULL && root->right->type == Terminal && root->right->symbol == '0')
            {
                //(f-0)->f
                return root->left;
            }
        }
        else if (root->symbol == '*')
        {
            if (root->left!=NULL && root->left->type == Terminal && root->left->symbol == '1')
            {
                //(1*f)->f
                return root->right;
            }
            if (root->right!=NULL && root->right->type == Terminal && root->right->symbol == '1')
            {
                //(f*1)->f
                return root->left;
            }
            if (root->left!=NULL && root->left->type == Terminal && root->left->symbol == '0')
            {
                //(0*f)->0
                return create_node(Terminal, '0');
            }
            if (root->right!=NULL && root->right->type == Terminal && root->right->symbol == '0')
            {
                //(f*0)->0
                return create_node(Terminal, '0');
            }
        }
    }
    return root;
```

Функція simplify спрощує дерево за заданими правилами: (f+0), (0+f), (f-0), (f*1),

(1*f) → f, (f*0), (0*f) → 0. Функція спочатку рекурсивно спрощує піддерева, потім перевіряє чи є нода знаком, і якщо так перевіряє чи піддерева попадають під один з описаних випадків.

```
Node* transform(Node* root)
{
    if (root == NULL)
        return NULL;

    //перетворюємо ліве і праве піддерево
    root->left = transform(root->left);
    root->right = transform(root->right);

    //якщо це знак, перевіряємо чи формула попадає під одне з правил
    if (root->type == Sign)
    {
        //(f1+f2)*f3 -> (f1*f3)+(f2*f3)
        if (root->symbol == '*' && root->left!=NULL && root->left->type == Sign && root->left->symbol == '+' && root->right!=NULL)
        {
            Node *f1 = root->left->left;
            Node *f2 = root->left->right;
            Node *f3 = root->right;
            //створюємо (f1*f3)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
            part1->right = f3;
            //створюємо (f2*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f2;
            part2->right = f3;
            //(f1*f3)+(f2*f3)
            Node *combined = create_node(Sign, '+');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }
```

```
        //(f1-f2)*f3->(f1*f3)-(f2*f3)
        if (root->symbol == '*' && root->left!=NULL && root->left->type == Sign && root->left->symbol == '-' && root->right!=NULL)
        {
            Node *f1 = root->left->left;
            Node *f2 = root->left->right;
            Node *f3 = root->right;
            //створюємо (f1*f3)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
            part1->right = f3;
            //створюємо (f2*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f2;
            part2->right = f3;
            //створюємо (f1*f3)-(f2*f3)
            Node *combined = create_node(Sign, '-');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }
```

```
        //(f1*(f2+f3))->(f1*f2)+(f1*f3)
        if (root->symbol == '*' && root->right!=NULL && root->right->type == Sign && root->right->symbol == '+')
        {
            Node *f1 = root->left;
            Node *f2 = root->right->left;
            Node *f3 = root->right->right;
            //створюємо (f1*f2)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
            part1->right = f2;
            //створюємо (f1*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f1;
            part2->right = f3;
            //створюємо (f1*f2)+(f1*f3)
            Node *combined = create_node(Sign, '+');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }
```

```
        //(f1*(f2-f3))->(f1*f2)-(f1*f3)
        if (root->symbol == '*' && root->right!=NULL && root->right->type == Sign && root->right->symbol == '-')
        {
            Node *f1 = root->left;
            Node *f2 = root->right->left;
            Node *f3 = root->right->right;
            //створюємо (f1*f2)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
            part1->right = f2;
            //створюємо (f1*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f1;
            part2->right = f3;
            //створюємо (f1*f2)-(f1*f3)
            Node *combined = create_node(Sign, '-');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }
    }
    return root;
}
```

Функція transform перетворює дерево за заданими правилами:

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

(f1 - f2)*f3 ---> (f1 * f3) - (f2 * f3)

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

(f1*(f2 - f3)) ---> (f1 * f2) - (f1 * f3)

Функція спочатку рекурсивно трансформує піддерева, потім перевіряє чи є нода знаком, і якщо так перевіряє чи піддерева попадають під умови. Якщо попадають під одну з умов, створює нові піддерева і заміняє вказівник ноди на новий.

```
//створення нового вузла
Node* create_node(NodeType type, char value)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->type = type;
    node->symbol = value;
    node->left = node->right = NULL;
    return node;
}
```

Функція create_Node створює нову ноду та повертає вказівник на неї.

```
//виводить дерево у вигляді формули
void print_tree(Node* root)
{
    if (root == NULL) return;
    //якщо термінал - виводить символ
    if (root->type == Terminal)
    {
        printf("%c", root->symbol);
    }
    //якщо знак - спочатку рекурсивно виводить ліве піддерево, знак, праве дерево
    else
    {
        printf("(");
        print_tree(root->left);
        printf(" %c ", root->symbol);
        print_tree(root->right);
        printf(")");
    }
}
```

Функція print_tree виводить бінарне дерево у вигляді формули.

```
//пропустити пробели
void skip_spaces(const char** str)
{
    while (**str == ' ')
        (*str)++;
}
```

Допоміжна функція, пропускає пробели в строці.

```
//парсинг терминалу(цифра або літера)
Node* parse_term(const char** str)
{
    skip_spaces(str);
    if (isdigit(**str)||isalpha(**str))
    {
        Node* node = create_node(Terminal, **str);
        (*str)++;
        return node;
    }
    else
    {
        printf("Mistake: waiting for a terminal but got the '%c'\n", **str);
        exit(1);
    }
}
```

Функція parse_term перевіряє чи елемент строки є цифрою або літерою. Якщо так створює нову ноду з цим символом і повертає вказівник на нову ноду, якщо ні виводить повідомлення про помилку.

```
//парсинг формули
Node* parse_formula(const char** str)
{
    skip_spaces(str);
    //якщо формула починається з дужки
    if (**str == '(')
    {
        //пропустити дужку
        (*str)++;
        //рекурсивно ліву частину формули
        Node* left = parse_formula(str);

        skip_spaces(str);

        //зчитати знак
        if (**str == '+' || **str == '-' || **str == '*')
        {
            char operator = **str;
            Node* root = create_node(Sign, operator);
            (*str)++;
            //ліва частина дерева
            root->left = left;
            skip_spaces(str);
            //права частина дерева
            root->right = parse_formula(str);
            skip_spaces(str);
```

```
            if (**str == ')')
            {
                (*str)++;
                return root;
            }
            else
            {
                printf("Mistake: waiting for a ')'\n");
                exit(1);
            }
        }
        else
        {
            printf("Mistake: waited for an sign, but got the '%c'\n", **str);
            exit(1);
        }
    }

    //якщо не починається з дужки - повиннен бути термінал
    return parse_term(str);
}
```

Функція parse_formula перетворює строку на бінарне дерево. Формула або має схему (формула знак формула) або є простим терміналом. Якщо починається на дужку, перевіряє, що ліва частина правильна, що є знак. Якщо це правильно створює нову ноду зі знаком, додає ліву частину формули в ліве піддерево, перевіряє що права частина формули правильна, та що формула закінчується на дужку. Інакше, виводить повідомлення про помилку.

```
//обробляємо формули з файлу
void process_formulas(const char* filename)
{
    FILE* file = fopen(filename, "r");
    if (file==NULL)
    {
        perror("coulnd't open the file");
        exit(1);
    }
    char line[256];
    while (fgets(line, sizeof(line), file)!=NULL)
    {
        const char* cursor = line;
        Node* root = parse_formula(&cursor);

        printf("Original formula: ");
        print_tree(root);
        printf("\n");

        root = simplify(root);
        printf("Simplified formula: ");
        print_tree(root);
        printf("\n");
```

```
        root = transform(root);
        printf("Transformed formula: ");
        print_tree(root);
        printf("\n");

        free_tree(root);
        printf("----------------------------\n");
    }

    fclose(file);
}
```

Функція process_formulas обробляє формули з файлу. Відкриває файл, читає строку, перетворює строку на дерево, виводить оригінальне дерево, спрощене і трансформоване.

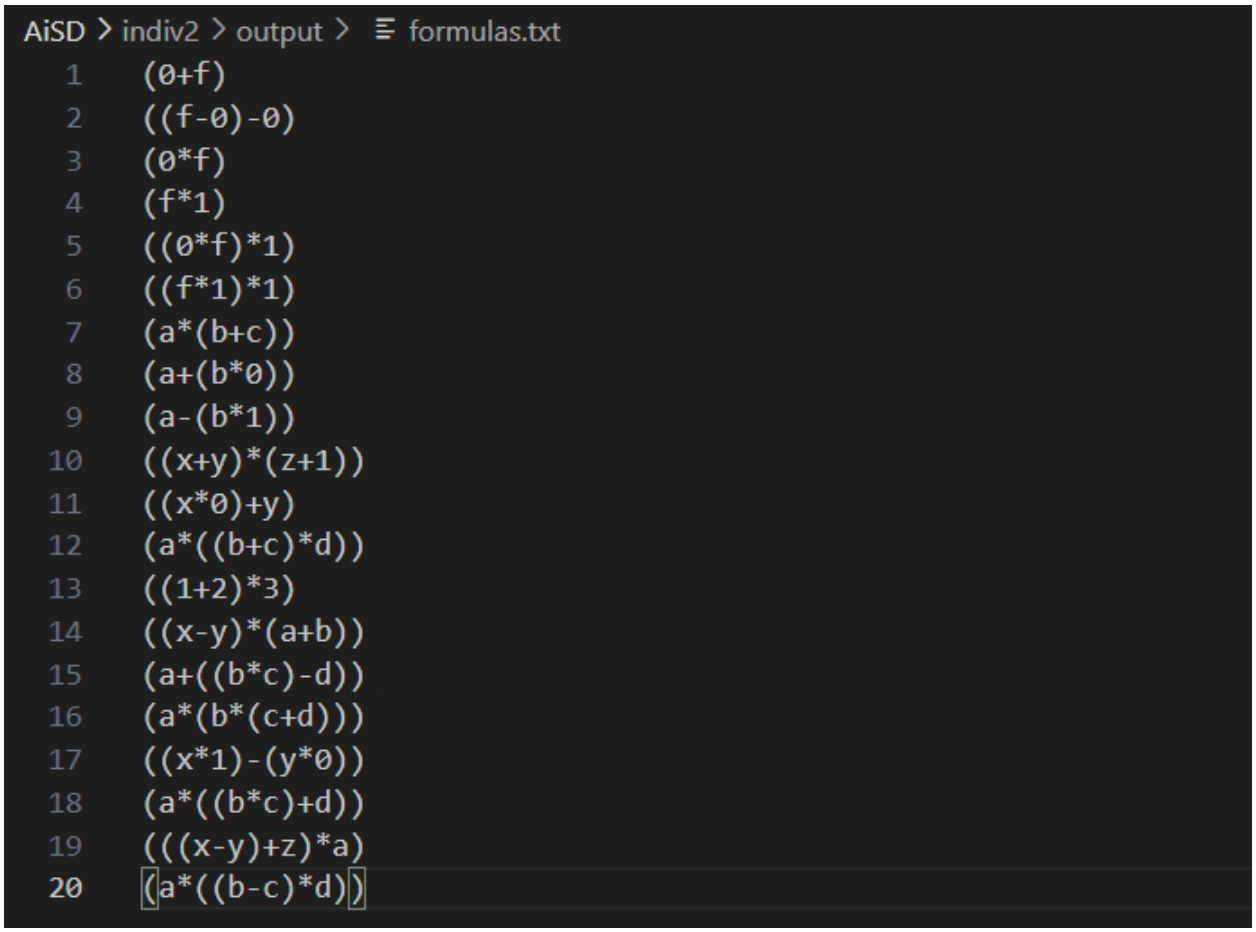
```
//рекурсивно звільнює пам'ять для дерева
void free_tree(Node* root)
{
    if (root == NULL) return;
    free_tree(root->left);
    free_tree(root->right);
    free(root);
}
```

Функція free_tree рекурсивно звільняє пам’ять для дерева.

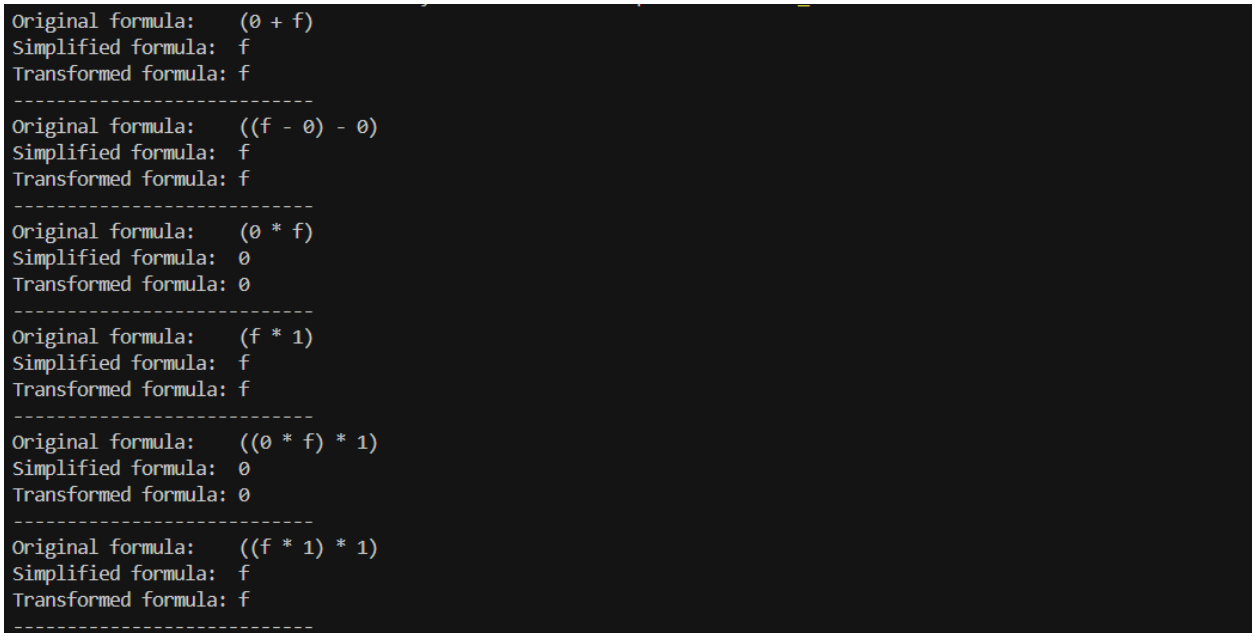
```
int main()
{
    const char* filename = "formulas.txt";
    process_formulas(filename);
    return 0;
}
```

Функція main викликає функцію process_formulas для файлу formulas.txt.

Приклади роботи програми:



Файл formulas.txt



```
----------------------------
Original formula:    (a * (b + c))
Simplified formula:  (a * (b + c))
Transformed formula: ((a * b) + (a * c))
----------------------------
Original formula:    (a + (b * 0))
Simplified formula:  a
Transformed formula: a
----------------------------
Original formula:    (a - (b * 1))
Simplified formula:  (a - b)
Transformed formula: (a - b)
----------------------------
Original formula:    ((x + y) * (z + 1))
Simplified formula:  ((x + y) * (z + 1))
Transformed formula: ((x * (z + 1)) + (y * (z + 1)))
----------------------------
Original formula:    ((x * 0) + y)
Simplified formula:  y
Transformed formula: y
----------------------------
Original formula:    (a * ((b + c) * d))
Simplified formula:  (a * ((b + c) * d))
Transformed formula: ((a * (b * d)) + (a * (c * d)))
----------------------------
Original formula:    ((1 + 2) * 3)
Simplified formula:  ((1 + 2) * 3)
Transformed formula: ((1 * 3) + (2 * 3))
----------------------------
Original formula:    ((x - y) * (a + b))
Simplified formula:  ((x - y) * (a + b))
Transformed formula: ((x * (a + b)) - (y * (a + b)))
----------------------------
Original formula:    (a + ((b * c) - d))
Simplified formula:  (a + ((b * c) - d))
Transformed formula: (a + ((b * c) - d))
----------------------------
```

```
----------------------------
Original formula:    (a * (b * (c + d)))
Simplified formula:  (a * (b * (c + d)))
Transformed formula: ((a * (b * c)) + (a * (b * d)))
----------------------------
Original formula:    ((x * 1) - (y * 0))
Simplified formula:  x
Transformed formula: x
----------------------------
Original formula:    (a * ((b * c) + d))
Simplified formula:  (a * ((b * c) + d))
Transformed formula: ((a * (b * c)) + (a * d))
----------------------------
Original formula:    (((x - y) + z) * a)
Simplified formula:  (((x - y) + z) * a)
Transformed formula: (((x - y) * a) + (z * a))
----------------------------
Original formula:    (a * ((b - c) * d))
Simplified formula:  (a * ((b - c) * d))
Transformed formula: ((a * (b * d)) - (a * (c * d)))
----------------------------
```

Вивід програми

Оригинальний код програми:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

typedef enum
{
    Sign, Terminal
}NodeType;

typedef struct Node
{
    NodeType type;
    char symbol;
    struct Node* left;
    struct Node* right;
} Node;

Node* simplify(Node* root);
Node* transform(Node* root);
Node* create_node(NodeType type, char value);
void skip_spaces(const char** str);
Node* parse_term(const char** str);
Node* parse_formula(const char** str);
void print_tree(Node* root);
void free_tree(Node* root);
void process_formulas(const char* filename);

int main()
{
    const char* filename = "formulas.txt";
    process_formulas(filename);
    return 0;
}



//спрощення дерева за правилами:(f+0), (0+f), (f-0), (f*1), (1*f), (f*0), (0*f)
```

```
Node* simplify(Node* root)
{
    if (root == NULL)
        return NULL;

    //спрощуємо праве та ліве піддерево
    root->left = simplify(root->left);
    root->right = simplify(root->right);

    //якщо це знак, то перевіряємо знак та піддерева
    if (root->type == Sign)
    {
        if (root->symbol == '+')
        {
            if (root->left!=NULL && root->left->type == Terminal &&
root->left->symbol == '0')
            {
                //(0+f)->f
                return root->right;
            }
            if (root->right!=NULL && root->right->type == Terminal &&
root->right->symbol == '0')
            {
                //(f+0)->f
                return root->left;
            }
        }
        else if (root->symbol == '-')
        {
            if (root->right!=NULL && root->right->type == Terminal &&
root->right->symbol == '0')
            {
                //(f-0)->f
                return root->left;
            }
        }
        else if (root->symbol == '*')
        {
```

```
            if (root->left!=NULL && root->left->type == Terminal &&
root->left->symbol == '1')
            {
                //(1*f)->f
                return root->right;
            }
            if (root->right!=NULL && root->right->type == Terminal &&
root->right->symbol == '1')
            {
                //(f*1)->f
                return root->left;
            }
            if (root->left!=NULL && root->left->type == Terminal &&
root->left->symbol == '0')
            {
                //(0*f)->0
                return create_node(Terminal, '0');
            }
            if (root->right!=NULL && root->right->type == Terminal &&
root->right->symbol == '0')
            {
                //(f*0)->0
                return create_node(Terminal, '0');
            }
        }
    }
    return root;
}

//перетворення дерева за заданними правилами:
//(f1 + f2)*f3 --->   (f1 * f3) + (f2 * f3)
//(f1 - f2)*f3 --->   (f1 * f3) - (f2 * f3)
//(f1*(f2 + f3)) ---> (f1 * f2) + (f1 * f3)
//(f1*(f2 - f3)) ---> (f1 * f2) - (f1 * f3)
Node* transform(Node* root)
{
    if (root == NULL)
        return NULL;
```

```
    //перетворюємо ліве і праве піддерево
    root->left = transform(root->left);
    root->right = transform(root->right);

    //якщо це знак, перевіряємо чи формула попадає під одне з правил
    if (root->type == Sign)
    {
        //(f1+f2)*f3 -> (f1*f3)+(f2*f3)
        if (root->symbol == '*' && root->left!=NULL && root->left->type == Sign
&& root->left->symbol == '+' && root->right!=NULL)
        {
            Node *f1 = root->left->left;
            Node *f2 = root->left->right;
            Node *f3 = root->right;
            //створюємо (f1*f3)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
            part1->right = f3;
            //створюємо (f2*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f2;
            part2->right = f3;
            //(f1*f3)+(f2*f3)
            Node *combined = create_node(Sign, '+');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }

        //(f1-f2)*f3->(f1*f3)-(f2*f3)
        if (root->symbol == '*' && root->left!=NULL && root->left->type == Sign
&& root->left->symbol == '-' && root->right!=NULL)
        {
            Node *f1 = root->left->left;
            Node *f2 = root->left->right;
            Node *f3 = root->right;
            //створюємо (f1*f3)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
```

```
            part1->right = f3;
            //створюємо (f2*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f2;
            part2->right = f3;
            //створюємо (f1*f3)-(f2*f3)
            Node *combined = create_node(Sign, '-');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }

        //(f1*(f2+f3))->(f1*f2)+(f1*f3)
        if (root->symbol == '*' && root->right!=NULL && root->right->type == Sign
&& root->right->symbol == '+')
        {
            Node *f1 = root->left;
            Node *f2 = root->right->left;
            Node *f3 = root->right->right;
            //створюємо (f1*f2)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
            part1->right = f2;
            //створюємо (f1*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f1;
            part2->right = f3;
            //створюємо (f1*f2)+(f1*f3)
            Node *combined = create_node(Sign, '+');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }

        //(f1*(f2-f3))->(f1*f2)-(f1*f3)
        if (root->symbol == '*' && root->right!=NULL && root->right->type == Sign
&& root->right->symbol == '-')
        {
            Node *f1 = root->left;
```

```
            Node *f2 = root->right->left;
            Node *f3 = root->right->right;
            //створюємо (f1*f2)
            Node *part1 = create_node(Sign, '*');
            part1->left = f1;
            part1->right = f2;
            //створюємо (f1*f3)
            Node *part2 = create_node(Sign, '*');
            part2->left = f1;
            part2->right = f3;
            //створюємо (f1*f2)-(f1*f3)
            Node *combined = create_node(Sign, '-');
            combined->left = part1;
            combined->right = part2;
            return combined;
        }
    }
    return root;
}

//створення нового вузла
Node* create_node(NodeType type, char value)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->type = type;
    node->symbol = value;
    node->left = node->right = NULL;
    return node;
}

//виводить дерево у вигляді формули
void print_tree(Node* root)
{
    if (root == NULL) return;
    //якщо термінал - виводить символ
    if (root->type == Terminal)
    {
        printf("%c", root->symbol);
    }
```

```
    //якщо знак - спочатку рекурсивно виводить ліве піддерево, знак, праве дерево
    else
    {
        printf("(");
        print_tree(root->left);
        printf(" %c ", root->symbol);
        print_tree(root->right);
        printf(")");
    }
}

//пропустити пробели
void skip_spaces(const char** str)
{
    while (**str == ' ')
        (*str)++;
}

//парсинг терминалу(цифра або літера)
Node* parse_term(const char** str)
{
    skip_spaces(str);
    if (isdigit(**str)||isalpha(**str))
    {
        Node* node = create_node(Terminal, **str);
        (*str)++;
        return node;
    }
    else
    {
        printf("Mistake: waiting for a terminal but got the '%c'\n", **str);
        exit(1);
    }
}

//парсинг формули
Node* parse_formula(const char** str)
{
    skip_spaces(str);
```

```
//якщо формула починається з дужки
if (**str == '(')
{
    //пропустити дужку
    (*str)++;
    //рекурсивно ліву частину формули
    Node* left = parse_formula(str);

    skip_spaces(str);

    //зчитати знак
    if (**str == '+' || **str == '-' || **str == '*')
    {
        char operator = **str;
        Node* root = create_node(Sign, operator);
        (*str)++;
        //ліва частина дерева
        root->left = left;
        skip_spaces(str);
        //права частина дерева
        root->right = parse_formula(str);
        skip_spaces(str);

        if (**str == ')')
        {
            (*str)++;
            return root;
        }
        else
        {
            printf("Mistake: waiting for a ')'\n");
            exit(1);
        }
    }
    else
    {
        printf("Mistake: waited for an sign, but got the '%c'\n", **str);
        exit(1);
    }
```

```
    }

    //якщо не починається з дужки - повиннен бути термінал
    return parse_term(str);
}

//обробляємо формули з файлу
void process_formulas(const char* filename)
{
    FILE* file = fopen(filename, "r");
    if (file==NULL)
    {
        perror("coulnd't open the file");
        exit(1);
    }
    char line[256];
    while (fgets(line, sizeof(line), file)!=NULL)
    {
        const char* cursor = line;
        Node* root = parse_formula(&cursor);

        printf("Original formula:    ");
        print_tree(root);
        printf("\n");

        root = simplify(root);
        printf("Simplified formula:  ");
        print_tree(root);
        printf("\n");

        root = transform(root);
        printf("Transformed formula: ");
        print_tree(root);
        printf("\n");

        free_tree(root);
        printf("----------------------------\n");
    }
```

```
    fclose(file);
}

//рекурсивно звільнює пам'ять для дерева
void free_tree(Node* root)
{
    if (root == NULL) return;
    free_tree(root->left);
    free_tree(root->right);
    free(root);
}
```