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

Студент группы М8О-103Б-22 Клименко Виталий Максимович, № по списку 11

Контакты www, e-mail, icq, skype vitalikklimenko96@gmail.com

Работа выполнена: 20 мая 2023 г.

Преподаватель: доцент Никулин С.П.

Входной контроль знаний с оценкой _____

Отчет сдан « » _____ 202 __ г., итоговая оценка ___

Подпись преподавателя _____

1. **Тема:** Деревья

2. **Цель работы:** Составить программу выполнения заданных преобразований арифметических выражений с применением деревьев

3. **Задание** (*вариант № 12*): Вынести общие сомножители из разности

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 4x 3.5GHz с ОП 16 ГБ НМД HDD 200 ГБ . Монитор Встроенный 1920x1080
Другие устройства Touchpad Synaptics

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 Pop!_OS версия 22.04 jammy
интерпретатор команд bash версия 5.1.16
Система программирования _____ версия _____
Редактор текстов _____ версия _____
Утилиты операционной системы _____
Прикладные системы и программы _____
Местонахождение и имена файлов программ и данных на домашнем компьютере _____

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

Составить структуру дерева: (Node)

char** tokens - список токенов для разбиения выражения на ветки дерева

char op - знак операции

char* constant - константа (переменная)

int64_t value - число

Node* left - левая ветка

Node* right - правая ветка

Node* prev - родитель

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

составить функции для обработки деревьев

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)

```
vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ ls
./  ../  l24-2012.djvu  main.c  makefile  misc.h  protocol.txt  report.pdf  tree.h

vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ cat makefile
CC = gcc
CFLAGS = -std=c99 -Wall -Wextra

main: main_o
        $(CC) $(CFLAGS) -o main.out main.o
main_o:
        $(CC) $(CFLAGS) -c main.c
clean:
        rm -f main.out *.o
vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ cat main.c
#include <stdio.h>

#define TREE_DEBUG 0
#define TREE_IMPLEMENTATION

#include "tree.h"


int main() {
    Node root;

    node_create_root(&root, "3*9 - 3 * 5 + 50");
    printf("Original:\n");
    node_build_tree(&root);
    node_print(root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    printf("\n");

    printf("Task:\n");
    node_take_out_factors(root.left->left, root.left->right);
    node_print(root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    printf("\n");

    printf("\n");

    node_create_root(&root, "3*9 + 50 - 3 * 5");
    printf("Original:\n");
    node_build_tree(&root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    node_print(root);
    printf("\n");

    printf("Task:\n");
    node_take_out_factors(root.left, root.right->right);
    node_print(root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    printf("\n");

    printf("\n");

    node_create_root(&root, "y*x - z * x + 50");
    printf("Original:\n");
    node_build_tree(&root);
    node_print(root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    printf("\n");

    printf("Task:\n");
    node_take_out_factors(root.left->left, root.left->right);
    node_print(root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    printf("\n");


    char expr[256] = {};
```

```
    printf("\nInput your math expression (without spaces): ");
    scanf("%s", &expr);

    node_create_root(&root, expr);
    printf("Original:\n");
    node_build_tree(&root);
    node_print(root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    printf("\n");

    printf("Task:\n");
    node_take_out_factors(root.left, root.right);
    node_print(root);
    printf("\nTree form:\n");
    node_print_tree(&root);
    printf("\n");

    return 0;
}
vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ cat tree.h
#ifndef TREE_H
#define TREE_H

#ifndef TREE_DEBUG
#define TREE_DEBUG 0
#endif

#include <inttypes.h>

typedef struct Node {
    char** tokens;
    char op;
    char* constant;
    int64_t value;
    struct Node* left;
    struct Node* right;
    struct Node* prev;
} Node;

void node_zero(Node* n);
void node_create_root(Node* n, char* lex);

void node_create_children(Node* n);
void node_build_tree(Node* n);

void node_print(Node n);
void node_print_tree(Node* n);
void node_print_debug(Node n);

void node_task(Node* n);
void node_take_out_factors(Node* l, Node* r);

#endif // TREE_H



#ifdef TREE_IMPLEMENTATION

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

#include "misc.h"

#define SEPARATOR "  "
#define TABS "\t\t\t\t\t\t\t\t\t\t"


void node_zero(Node* n) {
    n->tokens = NULL;
    n->op = ' ';
    n->constant = NULL;
    n->value = -1;
    n->left = NULL;
    n->right = NULL;
    n->prev = NULL;
}

void node_create_root(Node* n, char* lex) {
    node_zero(n);
    char** tokens = (char**) calloc(strlen(lex) + 1, sizeof(char*));
    tokens[strlen(lex)] = NULL;
```

```
    uint64_t t = 0;

    for (uint64_t i = 0; lex[i] != '\0'; ++i) {
        uint64_t j = 0;
        if (lex[i] == ' ') continue;

        if (char_is_operation(lex[i]) || char_is_bracket(lex[i])) {
            tokens[t] = (char*) calloc(1, sizeof(char));
            tokens[t][0] = lex[i];
            j = 1;
        } else {
            tokens[t] = (char*) calloc(10, sizeof(char));

            if ('0'<= lex[i] && lex[i] <= '9') {
                for (j = 0; lex[i + j] != '\0' && char_is_number(lex[i + j]); ++j);
            } else {
                for (j = 0; lex[i + j] != '\0' && !char_is_operation(lex[i + j]) &&
                            lex[i + j] != ' ' && !char_is_bracket(lex[i + j]); ++j);
            }

            memcpy(tokens[t], lex + i, j);
        }

        i += j - 1;
        ++t;
    }

    n->tokens = tokens;
}


void node_create_children(Node* n) {
    n->left = (Node *) calloc(1, sizeof(Node));
    n->right = (Node *) calloc(1, sizeof(Node));
    node_zero(n->left);
    node_zero(n->right);

    n->left->prev = n;
    n->right->prev = n;

    uint64_t length = 0, i = 0, j = 0;
    uint64_t lowest_priority_op_ind = 0, bracket_depth = 0, last_bracket_depth = 0;
    char op = ' ';

    for (length = 0; n->tokens[length] != NULL; ++length) {
        if (TREE_DEBUG) printf("LEN = %I64d\n", length);
        char cop = n->tokens[length][0];

        if (cop == '(') {
            bracket_depth++;
            if (TREE_DEBUG) printf("+BD = %I64d\n", bracket_depth);
            continue;
        } else if (cop == ')') {
            bracket_depth--;
            if (TREE_DEBUG) printf("-BD = %I64d\n", bracket_depth);
            continue;
        }

        if (bracket_depth > last_bracket_depth) continue;

        if (char_is_operation(cop)) {
            last_bracket_depth = bracket_depth;
            if (
                op == ' ' || op == '^' ||
                (op == '/' && cop == '*') ||
                ((op == '*' || op == '/') && (cop == '+' || cop == '-')) ||
                (op == '-' && cop == '+')
            ) {
                op = cop;
                lowest_priority_op_ind = length;
            }
        }
    }
    n->left->op = op;
    if (TREE_DEBUG) printf("OP: %c\n\n", n->left->op);

    uint64_t left_length = lowest_priority_op_ind;
    uint64_t right_length = length - left_length - 1;

    uint64_t left_braces = 0, right_braces = 0, cur_braces = 0, off = 0;
    for (i = 0; i < left_length/2; ++i) {
        if (n->tokens[i][0] == '(' && n->tokens[left_length - 1 - i][0] == ')') {
            left_braces++;
        } else break;
```

```
}

for (i = left_braces; i < left_length - left_braces; ++i) {
    if (n->tokens[i][0] == ')') {
        if (cur_braces == 0) {
            off++;
        } else {
            cur_braces--;
        }
    } else if (n->tokens[i][0] == '(') {
        cur_braces++;
    }
}

off += cur_braces;
left_braces -= off/2;
left_length -= 2*left_braces;


off = 0, cur_braces = 0;
for (i = 0; i < right_length; ++i) {
    if (n->tokens[left_length + 1 + 2 * left_braces + i][0] == '(' && n->tokens[length - 1 - i][0] == ')') {
        right_braces++;
    } else break;
}

for (i = right_braces; i < right_length - right_braces; ++i) {
    if (n->tokens[length - right_length + i][0] == ')') {
        if (cur_braces == 0) {
            off++;
        } else {
            cur_braces--;
        }
    } else if (n->tokens[length - right_length + i][0] == '(') {
        cur_braces++;
    }
}
off += cur_braces;
right_braces -= off/2;
right_length -= 2*right_braces;


if (TREE_DEBUG) printf("braces: %I64d, %I64d\n", left_braces, right_braces);
if (TREE_DEBUG) printf("lengths: %I64d, %I64d\n\n", left_length, right_length);

n->left->tokens = (char**) calloc(left_length + 1, sizeof(char*)); // one extra for NULL
n->right->tokens = (char**) calloc(right_length + 1, sizeof(char*));

n->left->tokens[left_length - 1] = NULL;
n->right->tokens[right_length - 1] = NULL;


uint64_t n_ind = 0;

uint64_t left_start = left_braces;
if (left_length == 1) {
    if (TREE_DEBUG) printf("GET THIS FOCKING LEFT MATE\n");
    n_ind = left_start;
    if (char_is_number(n->tokens[n_ind][0])) {
        n->left->value = atoll(n->tokens[n_ind]);
        if (TREE_DEBUG) printf("NUMBER! %I64d\n", n->left->value);
    } else {
        n->left->constant = n->tokens[n_ind];
        if (TREE_DEBUG) printf("CONST! %s\n", n->left->constant);
    }
} else {
    if (TREE_DEBUG) printf("COPY TO L:\n");
    for (i = 0; i < left_length; ++i) {
        n_ind = left_start + i;
        n->left->tokens[i] = (char*) calloc(strlen(n->tokens[n_ind]), sizeof(char));
        for (j = 0; j < strlen(n->tokens[n_ind]); ++j) {
            n->left->tokens[i][j] = n->tokens[n_ind][j];
        }
        if (TREE_DEBUG) printf("%s", n->left->tokens[i]);
    }
    if (TREE_DEBUG) printf("\n");
}
if (TREE_DEBUG) printf("\n");

uint64_t right_start = left_length + 1 + 2 * left_braces + right_braces;
if (right_length == 1) {
    if (TREE_DEBUG) printf("GET THIS FOCKING RIGHT MATE\n");
    n_ind = right_start;
    if (char_is_number(n->tokens[n_ind][0])) {
        n->right->value = atoll(n->tokens[n_ind]);
```

```
            if (TREE_DEBUG) printf("NUMBER! %I64d\n", n->right->value);
        } else {
            n->right->constant = n->tokens[n_ind];
            if (TREE_DEBUG) printf("CONST! %s\n", n->right->constant);
        }
    } else {
        if (TREE_DEBUG) printf("COPY TO R:\n");
        for (i = 0; i < right_length; ++i) {
            n_ind = right_start + i;
            n->right->tokens[i] = (char*) calloc(strlen(n->tokens[n_ind]), sizeof(char));
            for (j = 0; j < strlen(n->tokens[n_ind]); ++j) {
                n->right->tokens[i][j] = n->tokens[n_ind][j];
            }
            if (TREE_DEBUG) printf("%s", n->right->tokens[i]);
        }
        if (TREE_DEBUG) printf("\n");
    }
    if (TREE_DEBUG) printf("\n");
}

void node_build_tree(Node* n) {
    if (TREE_DEBUG) printf("[NET]\n");
    if (n->tokens == NULL) return;

    uint64_t length;
    for (length = 0; n->tokens[length] != NULL; ++length);
    if (length > 0) {
        if (TREE_DEBUG) printf("===CHID===\n");
        node_create_children(n);
        if (TREE_DEBUG) printf("===LEFT===\n");
        node_build_tree(n->left);
        if (TREE_DEBUG) printf("===RIHT===\n");
        node_build_tree(n->right);
    } else {
        if (TREE_DEBUG) printf("EMPTY!\n\n");
    }
}



void node_print(Node n) {
    if (n.left != NULL) {
        printf("(");
        node_print(*n.left);
        printf("%c", n.left->op);
    }

    if (n.constant == NULL) {
        if (n.value != -1) {
            printf("%I64d", n.value);
        }
    } else {
        printf("%s", n.constant);
    }

    if (n.right != NULL) {
        node_print(*n.right);
        printf(")");
    }
}

void _node_print_tree(Node* n, int lvl) {
    if (n == NULL) return;

    lvl += 1;
    int seps = (int) (strlen(SEPARATOR) * lvl);

    _node_print_tree(n->right, lvl);

    if (n->constant == NULL) {
        if (n->value != -1) {
            printf("%*s%I64d", (int) (seps - strlen(SEPARATOR)), SEPARATOR, n->value);
        }
    } else {
        printf("%*s%s", (int) (seps - strlen(SEPARATOR)), SEPARATOR, n->constant);
    }

    if (n->left != NULL) printf("%*s%c", seps, SEPARATOR, n->left->op);
    printf("\n");

    _node_print_tree(n->left, lvl);
}

void node_print_tree(Node* n) {
    _node_print_tree(n, 1);
```

```
}

void _node_print_debug(Node n, int lvl) {
    printf("Node {\n" "%.*stokens: ", lvl, TABS);
    if (n.tokens != NULL) {
        for (uint64_t i = 0; n.tokens[i] != NULL; ++i)
            printf("%s ", n.tokens[i]);
        printf("\n");
    } else {
        printf("(null)\n");
    }

    printf(
        "%.*sop: %c\n"
        "%.*sconstant: %s\n"
        "%.*svalue: %I64d\n"
        "%.*sleft: ",
        lvl, TABS, n.op,
        lvl, TABS, n.constant,
        lvl, TABS, n.value,
        lvl, TABS
    );

    if (n.left != NULL) {
        _node_print_debug(*n.left, lvl + 1);
    } else {
        printf("(null)\n");
    }

    printf("%.*sright: ", lvl, TABS);
    if (n.right != NULL) {
        _node_print_debug(*n.right, lvl + 1);
    } else {
        printf("(null)\n");
    }

    printf("%.*s}\n", lvl - 1, TABS);
}

void node_print_debug(Node n) {
    _node_print_debug(n, 1);
}

void node_task(Node* n) {
    // TODO: Find pairs of nodes that need to be processed
    // TODO: they should be connected only via + and -

    Node* l = n->left;
    Node* r = n->right;

    Node** left_nodes;
    Node** right_nodes;

    node_print_debug(*l);
    node_print_debug(*r);

    node_take_out_factors(l, r);
}

// move right part's multiplier to l and other stuff to r->prev
void node_take_out_factors(Node* l, Node* r) {
    int64_t llv = l->left->value, lrv = l->right->value;
    int64_t rlv = r->left->value, rrv = r->right->value;

    char *llc = l->left->constant, *lrc = l->right->constant;
    char *rlc = r->left->constant, *rrc = r->right->constant;

    int cmp_ll = 0, cmp_lr = 0, cmp_rl = 0, cmp_rr = 0;

    if (llc != NULL && rlc != NULL)
        cmp_ll = (strcmp(llc, rlc) == 0);

    if (llc != NULL && rrc != NULL)
        cmp_lr = strcmp(llc, rrc) == 0;

    if (lrc != NULL && rlc != NULL)
        cmp_rl = strcmp(lrc, rlc) == 0;

    if (lrc != NULL && rrc != NULL)
        cmp_rr = strcmp(lrc, rrc) == 0;


    if ( // checks for operations are done in node_task()
        !(
            ( // check if matching numbers are present
```

```
            ((llv != -1 && rlv != -1) && llv == rlv) ||
            ((llv != -1 && rrv != -1) && llv == rrv) ||
            ((lrv != -1 && rlv != -1) && lrv == rlv) ||
            ((lrv != -1 && rrv != -1) && lrv == rrv)
        )
        ||
        ( // check if matching constants are present
            cmp_ll || cmp_lr || cmp_rl || cmp_rr
        )
    )
) return;


Node* new_l = (Node*) calloc(1, sizeof(Node));
node_zero(new_l);

new_l->left = (Node*) calloc(1, sizeof(Node));
node_zero(new_l->left);
new_l->right = (Node*) calloc(1, sizeof(Node));
node_zero(new_l->right);

new_l->right->left = (Node*) calloc(1, sizeof(Node));
node_zero(new_l->right->left);
new_l->right->right = (Node*) calloc(1, sizeof(Node));
node_zero(new_l->right->right);


new_l->prev = l->prev;
if (l->prev == r->prev) {
    new_l->op = l->prev->op;
} else {
    new_l->op = l->op;
}
new_l->left->op = l->left->op;
new_l->right->left->op = r->prev->left->op;


if (llv != -1 && lrv != -1 && rlv != -1 && rrv != -1) {
    int64_t value1 = -1, value2 = -1;
    int64_t factor = -1;

    if (llv == rlv) {
        factor = llv;
        value1 = lrv;
        value2 = rrv;
    } else if (llv == rrv) {
        factor = llv;
        value1 = lrv;
        value2 = rlv;
    } else if (lrv == rlv) {
        factor = lrv;
        value1 = llv;
        value2 = rrv;
    } else {
        factor = lrv;
        value1 = llv;
        value2 = rlv;
    }

    new_l->left->value = factor;
    new_l->right->left->value = value1;
    new_l->right->right->value = value2;
} else if (cmp_ll || cmp_lr || cmp_rl || cmp_rr) {
    char *constant1 = NULL, *constant2 = NULL;
    char *factor = NULL;

    if (cmp_ll) {
        factor = llc;
        constant1 = lrc;
        constant2 = rrc;
    } else if (cmp_lr) {
        factor = llc;
        constant1 = lrc;
        constant2 = rlc;
    } else if (cmp_rl) {
        factor = lrc;
        constant1 = llc;
        constant2 = rrc;
    } else {
        factor = lrc;
        constant1 = llc;
        constant2 = rlc;
    }

    new_l->left->constant = factor;
```

```
        new_l->right->left->constant = constant1;
        new_l->right->right->constant = constant2;
    }

    if (l->prev == r->prev) {
        *l->prev = *new_l;
    } else {
        *l = *new_l;
        Node* temp = r->prev->left;
        temp->op = ' ';
        temp->prev = r->prev->prev;
        r->prev->prev->right = temp;
    }
}

#endif // TREE_IMPLEMENTATION
vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ cat misc.h
#ifndef MISC_H
#define MISC_H


int char_is_operation(char c) {
    return c == '+' || c == '-' || c == '*' || c == '/' || c == '^';
}

int char_is_number(char c) {
    return '0' <= c && c <= '9';
}

int char_is_bracket(char c) {
    return c == '(' || c == ')';
}

#endif
vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ make
gcc -std=c99 -Wall -Wextra -c main.c
In file included from main.c:6:
tree.h: In function 'node_task':
tree.h:370:12: warning: unused variable 'right_nodes' [-Wunused-variable]
     Node** right_nodes;
            ^~~~~~~~~~~
tree.h:369:12: warning: unused variable 'left_nodes' [-Wunused-variable]
     Node** left_nodes;
            ^~~~~~~~~~
main.c: In function 'main':
main.c:64:13: warning: format '%s' expects argument of type 'char *', but argument 2 has type 'char (*)[256]' [-Wformat=]
     scanf("%s", &expr);
             ~^   ~~~~~
main.c:64:13: warning: format '%s' expects argument of type 'char *', but argument 2 has type 'char (*)[256]' [-Wformat=]
gcc -std=c99 -Wall -Wextra -o main.out main.o

vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ ./main.out
Original:
(((3*9)-(3*5))+50)
Tree form:
    50
    +
        5
        *
        3
      -
        9
        *
        3

Task:
((3*(9-5))+50)
Tree form:
    50
    +
        5
        -
        9
      *
      3


Original:

Tree form:
        5
        *
```

```
        3
      -
      50
    +
      9
      *
      3
((3*9)+(50-(3*5)))
Task:
((3*(9-5))+50)
Tree form:
    50
    +
        5
        -
        9
      *
      3


Original:
(((y*x)-(z*x))+50)
Tree form:
    50
    +
        x
        *
        z
      -
        x
        *
        y

Task:
((x*(y-z))+50)
Tree form:
    50
    +
        z
        -
        y
      *
      x


Input your math expression (without spaces): a*b-c*b
Original:
((a*b)-(c*b))
Tree form:
      b
      *
      c
    -
      b
      *
      a

Task:
(b*(a-c))
Tree form:
      c
      -
      a
    *
    b


vital@vitos-hp16 MINGW64 /c/important/docs/mai/labs/l24 (master)
$ ./main.out
Original:
(((3*9)-(3*5))+50)
Tree form:
    50
    +
        5
        *
        3
      -
        9
        *
        3

Task:
((3*(9-5))+50)
Tree form:
```

```
    50
    +
        5
        -
        9
      *
      3


Original:

Tree form:
        5
        *
        3
      -
      50
    +
      9
      *
      3
((3*9)+(50-(3*5)))
Task:
((3*(9-5))+50)
Tree form:
    50
    +
        5
        -
        9
      *
      3


Original:
(((y*x)-(z*x))+50)
Tree form:
    50
    +
        x
        *
        z
      -
        x
        *
        y

Task:
((x*(y-z))+50)
Tree form:
    50
    +
        z
        -
        y
      *
      x


Input your math expression (without spaces): 1*14-15*14
Original:
((1*14)-(15*14))
Tree form:
      14
      *
      15
    -
      14
      *
      1

Task:
(14*(1-15))
Tree form:
      15
      -
      1
    *
    14
```

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: Не получилось составить функцию, с помощью которой можно было бы находить подвыражения, которые можно обработать данных алгоритмом, так же изначальная структура дерева задумывалась для обработки вообще любых математических выражений, а не простых выражений, у которых числа не больше 9 и однобуквенные переменные как в книге Валентина Евгеньевича, из-за чего код оказался громоздким и вообще лабораторная работа слишком затянулась

11. **Выводы:** Я научился работать со структурой данных дерево. Больше узнал о реализации нелинейных структур данных. Научился писать парсеры.

Недочёты при выполнении задания могут быть устранены следующим образом: _____

Подпись студента _____