МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №1

по дисциплине: Математическая логика и теория алгоритмов
тема: «Логика высказываний»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили: ст. пр. Бондаренко Татьяна
Владимировна

Белгород 2023 г.

Цель работы: решить задачи с использованием логических высказываний, разработать программу для нахожданиея значение формулы, представленной в ДНФ или КНФ на данной интерпретации.
Вариант № 10

Задания к работе:

1. Решение предложенных в теоретической части задач.
2. Программа на выбранном языке программирования в виде исходных кодов (с поясняющими комментариями) и в электронном варианте для демонстрации на ЭВМ.
3. Спецификация программы с указанием основных структур данных и алгоритмов.
4. Наборы тестовых данных.

Исходный код *log_calculator.h*

```c
#ifndef MATH_LOGIC_LOG_CALCULATOR_H
#define MATH_LOGIC_LOG_CALCULATOR_H

#include <stdbool.h>

#define INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM 0
#define INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM 1
#define INPUT_TYPE_INVALID (-1)

#define LITERAL_POSITIVE 1
#define LITERAL_UNDEF 0
#define LITERAL_NEGATIVE (-1)

#define LATIN_ALPHABET_LENGTH 26

#define NON '!'
#define CONJUNCTION '&' // Стрелочка вверх
#define DISJUNCTION '+' // Стрелочка вниз
#define OPENING_BRACKET '('
#define CLOSING_BRACKET ')'
#define charAlphabetIndex(val) ((val) - 'A')

typedef struct Formula {
    // Элементы формулы, конъюнкты или дизъюнкты в зависимости от type
    char **val;
    // Длина массива элементов
    int amount;
    // Вместимость массива элементов
    int capacity;
    // Тип формулы
    int type;
} Formula;

// Принимает тип формулы type, возвращает INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM, если переданный тип
↪  соответствует дизъюнктивной нормальной форме,
// INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM, если переданный тип соответствует конъюнктивной нормальной
↪  форме, иначе - INPUT_TYPE_INVALID.
int checkInputType(int type);

// Принимает тип формулы type и на его основе инициализирует и возвращает формулу.
// Если type невалидный, возвращает формулу с типом INPUT_TYPE_INVALID.
```

```
Formula initFormula(int type);

// Освобождает из памяти ресурсы, занятые formula
void freeFormula(Formula *formula);

// Добавляет element (конъюнкт или дизъюнкт в зависимости от типа формулы) в formula.
void addElement(Formula *formula, char element[LATIN_ALPHABET_LENGTH]);

// Выполняет парсинг формулы и сохраняет её в formula из строки line.
// Формула в конъюнктивной нормальной форме должна иметь вид "(... + ... + ...) & (... + ... +
↪  ... + ...) & (... + ... + ...)"
// Формула в дизъюнктивной нормальной форме должна иметь вид "... & ... & ... + ... & ... & ... &
↪  ... + ... & ... & ..."
// Возвращает false, если парсинг был выполнен успешно, иначе - true.
bool processFormula(Formula *formula, char* line);

// Возвращает значение формулы f при значениях переменных из val.
bool findVal(Formula f, bool val[LATIN_ALPHABET_LENGTH]);

#endif //MATH_LOGIC_LOG_CALCULATOR_H
```

log_calculator.c

```
#include <malloc.h>
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include <stdbool.h>

#include "log_calculator.h"

int checkInputType(int type)
{
    // Если тип соответствует существующим возможным типам, возращаем этот тип.
    if (type == INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM || type == INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM)
        return type;

    // Иначе - невалидный тип
    return INPUT_TYPE_INVALID;
}

Formula initFormula(int type)
{
```

```
    // Если тип невалидный - возвращаем формулу с ошибочным типом
    if (checkInputType(type) == INPUT_TYPE_INVALID)
        return (Formula){NULL, 0, 0, INPUT_TYPE_INVALID};
    return (Formula){malloc(0), 0, 0, type};
}

void freeFormula(Formula *formula)
{
    // Удаляем каждый элемент формулы
    for (int i = 0; i < formula->amount; i++)
        free(formula->val[i]);

    // Сам массив формул
    free(formula->val);

    // Делаем формулу "невалидной"
    formula->val = NULL;
    formula->amount = 0;
    formula->capacity = 0;
    formula->type = INPUT_TYPE_INVALID;
}

void addElement(Formula *formula, char element[LATIN_ALPHABET_LENGTH])
{
    // Если количество больше вместимости, увеличиваем размер массива элементов в 2 раза + 1
    if (formula->amount >= formula->capacity)
        formula->val = realloc(formula->val, formula->capacity = (formula->capacity * 2 + 1));

    // Инициализируем массив переменных элемента формулы
    formula->val[formula->amount++] = malloc(sizeof(char) * LATIN_ALPHABET_LENGTH);
    // Копируем переменные в элемент формулы
    memcpy(formula->val[formula->amount - 1], element, sizeof(char) * LATIN_ALPHABET_LENGTH);
}

static bool _processFormulaDisjunctive(Formula *formula, char *line)
{
    // ... & ... & ... + ... & ... & ... & ... + ... & ... & ...

    int presence = LITERAL_UNDEF;
    int letterIndex = -1;
    char element[LATIN_ALPHABET_LENGTH] = {};
    int index = 0;
    while (1)
```

```
{
    char input = *line;
    if (isspace(input))
    {
        // Игнорируем, таким образом пробелы, табуляции и т.д. не будут иметь значения.
    }
    else if (input == NON)
    {
        // Отрицание должно стоять до переменной, а не после.
        if (letterIndex != -1)
        {
            fprintf(stderr, "Parsing error at index %d\n", index);
            return 1;
        }

        // Если отрицаний не было, значит переменная стоит с знаком !.
        // Если отрицание было, то !! = отсутствие отрицаний.
        presence = presence == LITERAL_UNDEF ? LITERAL_NEGATIVE : presence ==
        ↪  LITERAL_POSITIVE ? LITERAL_NEGATIVE
                                                                                ↪
                                                                                ↪


    }
    else if (input == CONJUNCTION)
    {
        // Исключение ситуации, подобной "&&".
        if (letterIndex == -1)
        {
            fprintf(stderr, "No literal present at index %d\n", index);
            return 1;
        }

        // Если встречаем &, сохраняем предыдущую переменную в элемент
        element[letterIndex] = presence == LITERAL_UNDEF ? LITERAL_POSITIVE : presence;
        presence = LITERAL_UNDEF;
        letterIndex = -1;
    }
    else if (input == DISJUNCTION)
    {
        // Исключение ситуации, подобной "++".
        if (letterIndex == -1)
        {
            fprintf(stderr, "No literal present at index %d\n", index);
```

```
            return 1;
        }

        // Если встречаем +, сохраняем предыдущую переменную в элемент
        element[letterIndex] = presence == LITERAL_UNDEF ? LITERAL_POSITIVE : presence;
        presence = LITERAL_UNDEF;
        letterIndex = -1;

        // А также сам элемент в формулу. Также обновляем буфферный элемент element.
        addElement(formula, element);
        memset(element, LITERAL_UNDEF, LATIN_ALPHABET_LENGTH);

    }
    else if (isupper(input) || islower(input))
    {
        // Найдена буква латинского алфавита. Выполняем дополниительные проверки и
        ↪  преобразования так, чтобы 'x' == 'X'.
        letterIndex = charAlphabetIndex(islower(input) ? toupper(input) : input);
    }
    else if (input == '\0')
    {
        // Если дошли до конца строки, сохраняем переменную в элемент, элемент в формулу.
        if (letterIndex == -1)
        {
            fprintf(stderr, "No literal present at index %d\n", index);
            return 1;
        }

        element[letterIndex] = presence == LITERAL_UNDEF ? LITERAL_POSITIVE : presence;

        addElement(formula, element);
        memset(element, LITERAL_UNDEF, LATIN_ALPHABET_LENGTH);

        // Заканчиваем цикл чтения.
        break;
    }
    else
    {
        fprintf(stderr, "Unknown character '%c' at index %d\n", input, index);
        return 1;
    }

    line++, index++;
```

```
    }

    return 0;
}

static bool _processFormulaConjunctive(Formula *formula, char *line)
{
    // (... + ... + ...) & (... + ... + ... + ...) & (... + ... + ...)

    int bracesCount = 0;
    int presence = LITERAL_UNDEF;
    int letterIndex = -1;
    char element[LATIN_ALPHABET_LENGTH] = {};
    bool elementEmpty = true;
    int index = 0;
    while (1)
    {
        char input = *line;
        if (isspace(input))
        {
            // Игнорируем, таким образом пробелы, табуляции и т.д. не будут иметь значения.
        }
        else if (input == OPENING_BRACKET)
        {
            // Открывающая скобка возможна, если степень вложенности равна 0, она не содержится
            ↪ внутри конъюнкта, конъюнкт пустой.
            if (presence != LITERAL_UNDEF || letterIndex != -1 || bracesCount != 0 ||
            ↪ !elementEmpty)
            {
                fprintf(stderr, "Parsing error at index %d\n", index);
                return 1;
            }

            bracesCount++;
        }
        else if (input == CLOSING_BRACKET)
        {
            // Открывающая скобка возможна, если степень вложенности равна 1, а также конъюнкт не
            ↪ пустой.
            if (letterIndex == -1 || bracesCount != 1)
            {
                fprintf(stderr, "Parsing error at index %d\n", index);
                return 1;
```

```
        }

        // Сохраняем переменную в элемент.
        element[letterIndex] = presence == LITERAL_UNDEF ? LITERAL_POSITIVE : presence;
        presence = LITERAL_UNDEF;
        letterIndex = -1;
        bracesCount--;
        elementEmpty = false;
    }
    else if (input == NON)
    {
        // Отрицание должно стоять до переменной, а не после.
        if (letterIndex != -1)
        {
            fprintf(stderr, "Parsing error at index %d\n", index);
            return 1;
        }

        // Если отрицаний не было, значит переменная стоит с знаком !.
        // Если отрицание было, то !! = отсутствие отрицаний.
        presence = presence == LITERAL_UNDEF ? LITERAL_NEGATIVE : presence ==
        ↪  LITERAL_POSITIVE ? LITERAL_NEGATIVE



    }
    else if (input == CONJUNCTION)
    {
        // Встретили &. Если вложенность равна 0, записываем элемент в формулу.
        if (bracesCount == 0)
        {
            addElement(formula, element);
            memset(element, LITERAL_UNDEF, LATIN_ALPHABET_LENGTH);
            elementEmpty = true;
        }
        else
        {
            fprintf(stderr, "Brackets error at index %d\n", index);
            return 1;
        }
    }
    else if (input == DISJUNCTION)
    {
```

```
            // Встретили +. Сохраняем переменную в элемент.
            if (letterIndex == -1)
            {
                fprintf(stderr, "No literal present at index %d\n", index);
                return 1;
            }

            element[letterIndex] = presence == LITERAL_UNDEF ? LITERAL_POSITIVE : presence;
            presence = LITERAL_UNDEF;
            letterIndex = -1;
            elementEmpty = false;
        }
        else if (isupper(input) || islower(input))
        {
            // Найдена буква латинского алфавита. Выполняем дополниительные проверки и
            ↪ преобразования так, чтобы 'x' == 'X'.
            letterIndex = charAlphabetIndex(islower(input) ? toupper(input) : input);
        }
        else if (input == '\0')
        {
            // Дошли до конца строки. Записываем элемент в формулу. Проверяем, что степень
            ↪ вложенности == 0.
            if (bracesCount == 0)
            {
                addElement(formula, element);
                memset(element, LITERAL_UNDEF, LATIN_ALPHABET_LENGTH);
            }
            else
            {
                fprintf(stderr, "Brackets error at index %d\n", index);
                return 1;
            }

            // Заканчиваем цикл чтения.
            break;
        }
        else
        {
            fprintf(stderr, "Unknown character '%c' at index %d\n", input, index);
            return 1;
        }

        line++, index++;
```

```
    }

    return 0;
}

bool processFormula(Formula *formula, char *line)
{
    // В зависимости от типа формулы выполняем парсинг при помощи соответствующей функции.
    int type = checkInputType(formula->type);
    if (type == INPUT_TYPE_INVALID)
        return 1;

    if (type == INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM)
    {
        return _processFormulaDisjunctive(formula, line);
    }

    if (type == INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM)
    {
        return _processFormulaConjunctive(formula, line);
    }
}

static bool _findValConjunctive(Formula f, bool val[LATIN_ALPHABET_LENGTH]) {
    // (... + ... + ...) & (... + ... + ... + ...) & (... + ... + ...)
    // Вычисления выполняются по ленивой схеме.

    for (int i = 0; i < f.amount; i++)
    {
        char *element = f.val[i];
        // Предположим, что результат элемента ... + ... + ... - false.
        bool elementResult = false;

        for (int j = 0; j < LATIN_ALPHABET_LENGTH; j++)
        {
            // Если переменной нет в формуле, пропускаем её.
            if (element[j] == LITERAL_UNDEF)
                continue;

            // Вычисляем значение элемента.
            int current = (element[j] == LITERAL_POSITIVE) ? val[j] : !val[j];

            // Если значение переменной true - присваиваем всему элементу true и заканчиваем
            ↪  цикл, дальнейшее вычисление не нужно.
```

```
            if (current)
            {
                elementResult = true;
                break;
            }
        }

        // Если элемент формулы false, то формула вида ... & ... & ... тоже будет иметь результат
        ↪ false
        if (!elementResult)
            return false;
    }

    // Все элементы ... & ... & ... true, формула тоже true.
    return true;
}

static bool _findValDisjunctive(Formula f, bool val[LATIN_ALPHABET_LENGTH]) {
    // ... & ... & ... + ... & ... & ... & ... + ... & ... & ...
    // Вычисления выполняются по ленивой схеме.
    for (int i = 0; i < f.amount; i++)
    {
        char *element = f.val[i];
        // Предположим, что результат элемента ... & ... & ... - true.
        bool elementResult = true;

        for (int j = 0; j < LATIN_ALPHABET_LENGTH; j++)
        {
            // Если переменной нет в формуле, пропускаем её.
            if (element[j] == LITERAL_UNDEF)
                continue;

            // Вычисляем значение элемента.
            int current = (element[j] == LITERAL_POSITIVE) ? val[j] : !val[j];
            // Если значение переменной false - присваиваем всему элементу false и заканчиваем
            ↪ цикл, дальнейшее вычисление не нужно.
            if (!current)
            {
                elementResult = false;
                break;
            }
        }
```

```
        // Если элемент формулы true, то формула вида ... + ... + ... тоже будет иметь результат
        ↪  true
        if (elementResult)
            return true;
    }

    // Все элементы оказались false, возвращаем false.
    return false;
}

bool findVal(Formula formula, bool args[LATIN_ALPHABET_LENGTH])
{
    // В зависимости от типа формулы выполняем вычисление значения формулы.
    int type = checkInputType(formula.type);
    if (type == INPUT_TYPE_INVALID)
        return false;

    if (type == INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM)
    {
        return _findValDisjunctive(formula, args);
    }

    if (type == INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM)
    {
        return _findValConjunctive(formula, args);
    }

    return false;
}
```

main.c

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

#include "../libs/alg/lab1/log_calculator.h"

int main()
{
    int type = INPUT_TYPE_INVALID;
    while (checkInputType(type) == INPUT_TYPE_INVALID) {
        printf("Select normal form type (0 - disjunctive, 1 - conjunctive): ");
```

```
        scanf("%d", &type);
    }

    char buf[512];
    gets(buf);
    Formula f;
    do {
    printf("Enter formula: ");
        f = initFormula(type);

        gets(buf);
    } while(processFormula(&f, buf));

    bool isPresent[LATIN_ALPHABET_LENGTH] = {};
    for (int i = 0; buf[i] != '\0'; i++) {
        if (islower(buf[i]))
            isPresent[buf[i] - 'a'] = true;
        if (isupper(buf[i]))
            isPresent[buf[i] - 'A'] = true;
    }

    printf("Enter variables:\n");
    for (int i = 0; i < LATIN_ALPHABET_LENGTH; i++) {
        if (!isPresent[i]) continue;

        printf("%c ", i + 'A');
    }
    printf("\n");

    bool v[LATIN_ALPHABET_LENGTH];
    for (int i = 0; i < LATIN_ALPHABET_LENGTH; i++) {
        if (!isPresent[i]) continue;

        scanf("%d", v + i);
    }

    printf("Result: %d", findVal(f, v));

    return 0;
}
```

Тесты

check_input_type.c

```c
#include "../../src/libs/alg/lab1/log_calculator.h"

#include <assert.h>

int main() {
    assert(checkInputType(0) == INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM);
    assert(checkInputType(1) == INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM);
    assert(checkInputType(99) == INPUT_TYPE_INVALID);
}
```

init_formula.c

```c
#include <assert.h>
#include <string.h>
#include "../../src/libs/alg/lab1/log_calculator.h"

int main() {
    Formula formula = initFormula(INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM);
    assert(formula.val != NULL);
    assert(formula.amount == 0);
    assert(formula.capacity == 0);
    assert(formula.type == INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM);

    Formula formula2 = initFormula(INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM);
    assert(formula2.val != NULL);
    assert(formula2.amount == 0);
    assert(formula2.capacity == 0);
    assert(formula2.type == INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM);

    Formula formula3 = initFormula(31293);
    assert(formula3.val == NULL);
    assert(formula3.amount == 0);
    assert(formula3.capacity == 0);
    assert(formula3.type == INPUT_TYPE_INVALID);

    return 0;
}
```

add_element.c

```c
#include <assert.h>
#include <string.h>
#include "../../src/libs/alg/lab1/log_calculator.h"

int main() {
    Formula formula = initFormula(INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM);
    char element[LATIN_ALPHABET_LENGTH] = {LITERAL_POSITIVE, LITERAL_NEGATIVE, 99,
    ↪  LITERAL_NEGATIVE};

    addElement(&formula, element);
    addElement(&formula, element);

    assert(formula.amount == 2);
    assert(formula.capacity >= formula.amount);
    assert(!strncmp(formula.val[0], element, LATIN_ALPHABET_LENGTH * sizeof(char)));
    assert(!strncmp(formula.val[1], element, LATIN_ALPHABET_LENGTH * sizeof(char)));

    return 0;
}
```

conjunctive_formula.c

```c
#include "../../src/libs/alg/lab1/log_calculator.h"

#include <assert.h>

bool formula(bool val[LATIN_ALPHABET_LENGTH])
{
    return (!val[0] || val[1]) && (val[0] + !val[1]);
}

int main()
{
    Formula f = initFormula(INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM);
    processFormula(&f, "(!a + b) & (a + !b)");
    for (int i = 0; i < (2 << 26); i++)
    {
        bool val[LATIN_ALPHABET_LENGTH];

        for (int j = 0; j < LATIN_ALPHABET_LENGTH; j++)
        {
            val[j] = !!(i & (1 << j));
        }
```

```
        assert(findVal(f, val) == formula(val));
    }
}
```

conjunctive_formula_stress.c

```c
#include "../../src/libs/alg/lab1/log_calculator.h"

#include <assert.h>

bool formula(bool val[LATIN_ALPHABET_LENGTH])
{
    return (val[15] || val[17] || !val[8] || val[14] || val[16] || !val[4] || val[22] || val[20]
    ↪  || val[3] || !val[7] || !val[2]) && (val[4] || val[7] || !val[20] || val[21] || val[5] ||
    ↪  val[25] || val[18] || val[17] || val[3] || val[15] || val[11] || !val[1] || val[9] ||
    ↪  !val[2] || !val[8] || !val[22] || val[10] || !val[19] || !val[14] || val[23] || !val[13]
    ↪  || !val[12] || val[6]) && (!val[6] || !val[11] || !val[16] || val[4] || !val[18] ||
    ↪  val[24] || !val[15] || !val[5] || val[12] || val[9] || !val[14] || !val[25]) && (val[3]
    ↪  || !val[10] || val[20] || !val[6] || val[8] || val[9] || val[17] || val[13] || !val[5] ||
    ↪  !val[21] || !val[11] || val[16] || !val[7] || val[23] || val[14] || val[24] || !val[0] ||
    ↪  !val[2]);
}

int main()
{
    Formula f = initFormula(INPUT_TYPE_CONJUNCTIVE_NORMAL_FORM);
    processFormula(&f, "     (    !  !  !   !   !  ! p  +  !  !  R  +  !    !  !
    ↪  !     ! i  +  !   !   !   !  O +    !  !  Q  +   !    !  !  e  +   !  !
    ↪  W +    !   !  u  +  D  +   !   !    !    !  !  H +    !    !   !  C
    ↪  )  &   (    !  !   !   !    !    !  E  + h  +  !    !  !    !  !   !  !
    ↪  !    !  u  +  !    !   !  !   !  ! V +   !    !   !  !   !   !    ! f
    ↪  +   !  ! Z  +    !   !  !   !   !   !  S  +   !   !  !    !  !   !  r  +
    ↪  !    !    !   !   !  !   !  ! d  +  !   !   !  !    !  !    !   ! P  +
    ↪  !  !  L  +   !   !    !  !  ! B  +  J  +  !    !  !   !   !  C +  !  !
    ↪  !   !    !  !   !  I  +  !    !  !    !  !   !  !  w  +  !   !    !   ! k +
    ↪  !   !   !  !    !  !   ! T  +   !   !  !   !    !  O  +    !    ! x  +     !
    ↪  N +    !    !   !   !    !  M  +   !    !   !  !    !   ! g      )  &    (
    ↪  ! G  +    !    !   !  l +    !   !    !  !    !  q +  !   !   !   !   !
    ↪  ! e +  !    !   ! S  +    !   !  !   !   !    !  y  +    !   !   !    ! !
    ↪  P +   !   !  !   !   !    ! ! f  +   !  !   !    ! m  + !   ! J  +     !
    ↪  !     !    !  !  O +   !   !   !    !   !  !    !   !   !  z      )  &
    ↪  (   !   !  D  +   !   !  !   !   !    !  ! K  +  u  +  !   !   ! g +    !
    ↪  !    !   !    !   !   !  ! i  +  !    !    !   !   !   ! J  +   !    !
    ↪  !   !    !   !  R  +   !   !   !   ! n +  !  !    !  f  +    !  !   !   !
    ↪  !    !   !  v +  !    !   !   !   !    !  ! L +  Q  +  !   !   !    !
    ↪  !  H + X  +   !   !   !    ! O +  y  +    !    !   !    !   !  ! ! a  +   !
    ↪  !     !    !    !  C   )  ");
```

```
    for (int i = 0; i < (2 << 26); i++)
    {
        bool val[LATIN_ALPHABET_LENGTH];

        for (int j = 0; j < LATIN_ALPHABET_LENGTH; j++)
        {
            val[j] = !!(i & (1 << j));
        }

        assert(findVal(f, val) == formula(val));
    }
}
```

disjunctive_formula.c

```
#include "../../src/libs/alg/lab1/log_calculator.h"

#include <assert.h>

bool formula(bool val[LATIN_ALPHABET_LENGTH])
{
    return !val[0] || val[1];
}

int main()
{
    Formula f = initFormula(INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM);
    processFormula(&f, "!a + b");
    for (int i = 0; i < (2 << 26); i++)
    {
        bool val[LATIN_ALPHABET_LENGTH];

        for (int j = 0; j < LATIN_ALPHABET_LENGTH; j++)
        {
            val[j] = !!(i & (1 << j));
        }

        assert(findVal(f, val) == formula(val));
    }
}
```

disjunctive_formula_stress.c

```c
#include "../../src/libs/alg/lab1/log_calculator.h"

#include <assert.h>

bool formula(bool val[LATIN_ALPHABET_LENGTH])
{
    return !val[7] && val[12] && val[19] && val[1] && val[5] && val[10] && !val[8] && val[17] &&
    ↪  val[23] && val[3] && !val[20] && !val[4] && val[25] && val[2] || val[6] && !val[16] &&
    ↪  val[22] && !val[12] && !val[25] && !val[5] || !val[17] && val[18] && val[1] && val[11] &&
    ↪  val[21] && !val[2] && val[7] && !val[4] && !val[13] && !val[3] && val[8] && !val[15] &&
    ↪  val[9] && !val[23] && !val[22] && val[12] && val[0] && val[10] && val[24] && val[5] &&
    ↪  val[25] && !val[19] && val[20] && !val[6] || val[15] && !val[24] && val[5] && !val[21] &&
    ↪  !val[18] && !val[12] && !val[11] && val[23] && !val[14] && val[0] && val[16];
}

int main()
{
    Formula f = initFormula(INPUT_TYPE_DISJUNCTIVE_NORMAL_FORM);
    processFormula(&f, "  !     !   !   !    !     !     !   H &   !  !   !  !   !    !
    ↪  !   ! M &    !   !   !    !   !   !  T &   !   !    !  ! b & F  &    !   !   !
    ↪  !    !   !  K  &     !   !   !    !   !   !   !  i   &   !    !   R   & x  & d  &
    ↪  !   !     !    !    !  !   !    !    ! U  &    !  e & z &   !     !   C     +    !
    ↪  !     !    !  G &     !   !     !    !    !     !     !   !  !  Q  &   !    !
    ↪  !  !    !   !   !  !  w   &    !   !  !   !   !    !   !     !     ! M &   !
    ↪  !     !  !    !   !    !  Z   &   !  !    !  !   !   !    !   !  !   F      +
    ↪  !    !    ! r &  !  !   !    !  S  &   !   !    !   !   !  !  b &    !   !  l
    ↪  &    !  !  !   !   !    ! v &   !    !    !  !   !   !     ! c   &  !   !  !
    ↪  ! H  &   !   e &     !   !   !    !    !    !   !  n &   !     !    !     !
    ↪  !   !   !   !   ! D  &   !   !   !   ! I   &    !    !   !    !  !   !   !  p
    ↪  &  !   !   !   !   !    !   J   &    !    !   !   !   !  !   !   !   !  x &
    ↪  !   !   !    !    !    !    !   !   ! w &   !    !   !    !   M   &    !   !  A
    ↪  &    !  ! k   & y &   !    !   !   !    !   !  !   ! f   &    !    !   !   !  Z
    ↪  &   !     !   !   !   !  !   !   !   ! T   &   u &   !   g      +    !  !   !
    ↪  !    !   !   !    ! p   &   !    !   !  !   !   !     !  !  !  Y   &   !    !
    ↪  !     !    !   !     !    ! f   &    !  !    !   V   &  !  S  &  !   !   !   !   !
    ↪  !   !     !   !  m   &   !   !   !   !   !  l  &    !   !  x &    !  O   &   A
    ↪  &    !   !    !   !   !    !    !  !   Q          ");
    for (int i = 0; i < (2 << 26); i++)
    {
        bool val[LATIN_ALPHABET_LENGTH];
```

```
        for (int j = 0; j < LATIN_ALPHABET_LENGTH; j++)
        {
            val[j] = !!(i & (1 << j));
        }

        assert(findVal(f, val) == formula(val));
    }
}
```

Результат выполнения тестов:

```
Test project /github/workspace/build
    Start 1: lab1CheckInputTypeTest
1/8 Test #1: lab1CheckInputTypeTest ...........   Passed    0.00 sec
    Start 2: lab1InitFormula
2/8 Test #2: lab1InitFormula ..................   Passed    0.00 sec
    Start 3: lab1FreeFormula
3/8 Test #3: lab1FreeFormula ..................   Passed    0.00 sec
    Start 4: lab1AddElement
4/8 Test #4: lab1AddElement ...................   Passed    0.00 sec
    Start 5: lab1DisjunctiveFormula
5/8 Test #5: lab1DisjunctiveFormula ...........   Passed   16.55 sec
    Start 6: lab1DisjunctiveFormulaStress
6/8 Test #6: lab1DisjunctiveFormulaStress .....   Passed   19.40 sec
    Start 7: lab1ConjunctiveFormula
7/8 Test #7: lab1ConjunctiveFormula ...........   Passed   14.41 sec
    Start 8: lab1ConjunctiveFormulaStress
8/8 Test #8: lab1ConjunctiveFormulaStress .....   Passed   22.69 sec

100% tests passed, 0 tests failed out of 8

Total Test time (real) =  73.06 sec
```

Вывод: в ходе выполнения лабораторной работы решили задачи с использованием логических высказываний, разработали программу для нахожданиея значение формулы, представленной в ДНФ или КНФ на данной интерпретации.