**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра МОЭВМ**

# ОТЧЕТ

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**ТЕМА: ИЕРАРХИЧЕСКИЕ СПИСКИ**

Вариант № 16

Студент гр. 8304 Рыжиков А. В.
Преподаватель Фирсов К. В.

Санкт-Петербург
2019

## 1 Цель работы.

Дано логическое выражение в префиксной форме. Необходимо проверить его синтаксическую корректность.

Дополнение 2 – группа заданий 16-24.

Пусть выражение (логическое, арифметическое, алгебраическое*) представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме ( (<операция> <аргументы>) ), либо в постфиксной форме (<аргументы> <операция>) ). Аргументов может быть 1, 2 и более. Например (в префиксной форме): (+ a (* b (- c))) или (OR a (AND b (NOT c))).

В задании даётся один из следующих вариантов требуемого действия с выражением: *проверка синтаксической корректности*, *упрощение* (преобразование), *вычисление*.

Пример *упрощения*: (+ 0 (* 1 (+ a b))) преобразуется в (+ a b).

В задаче *вычисления* на входе дополнительно задаётся список значений переменных

$$( (x_1\ c_1)\ (x_2\ c_2)\ \dots\ (x_k\ c_k)\ ),$$

где $x_i$ – переменная, а $c_i$ – её значение (константа).

В индивидуальном задании указывается: тип выражения (возможно дополнительно - состав операций), вариант действия и форма записи. Всего 9 заданий.

---

* - здесь примем такую терминологию: в *арифметическое* выражение входят операции +, -, *, /, а в *алгебраическое* – +, -, * и дополнительно некоторые функции.

16) логическое, проверка синтаксической корректности, добавить 4-ую операцию (которая может принимать 2 аргумента), префиксная форма

*! Переменной назовём маленькую букву английского алфавита.*

*! Операция AND ,OR и -> определены для 1,2 и более переменных*

*!Операция NOT определена для одной переменной*

## 2 Описание программы

Программа решает поставленную задачу при помощи иерархических списков и рекурсии. Рекурсивное решение

следует из того, что количество аргументов может может быть 1, 2 и более, притом что само выражение может быть аргументом. Рекурсивно заполняем элементы иерархического списка. За основу для решения задачи берём структуры Node и Listю Пройдя иерархический список, убедимся, что выражение принадлежит к синтаксически верному, выведем положительный результат, иначе отрицательный.

## 3.1 Зависимости и объявление функций, структуры данных

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;
typedef struct List;
typedef struct Node;

typedef struct Node {
    string my_data;
    bool isAtom;
    Node *next;
    List *list;
};

typedef struct List {
    Node *head;
};


bool isOrImplication(string myString, int num);

bool isAnd(string myString, int count);

bool isNot(string myString, int count);

bool isArgument(string myString, int *count, Node *node);

bool isEquenceArguments(string myString, int *count, Node *node);

bool isLogicExpression(string myString, int *count, Node *node);
```

Описание структур данных.

Структура Node содержит указатель на следующий элемент, имеет флаг isAtom, если он положителен, то структура также хранит в себе информация в поле my_data, если же он отрицателен, то указатель list указывает на непустой список.

Структура List содержит голову списка.

### 3.2 Функция main.

Программа решает поставленную задачу при помощи рекурсии и иерархических структур, чтение происходит из файла test2.txt (также возможен ввод данных вручную) .

```
int main() {

    int your_choose = 0;

    cout << "If you want to enter data from a file, enter \'1\'\n";
    cout << "If you want to enter data manually, enter \'2\'\n";

    cin >> your_choose;

    if (your_choose==1) {
        ifstream fin;
        fin.open("test2.txt");

        if (fin.is_open()) {
            cout << "Reading from file:" << "\n";

            int super_count = 0;

            while (!fin.eof()) {

                super_count++;

                string str;
                getline(fin, str);

                cout << "test #" << super_count << " \"" + str + "\"" << "\n";
                check(str);
```

```
            }
        } else {
            cout << "File not opened";
        }

        fin.close();
    } else{
        if(your_choose==2){
            cout << "Enter data \n";
            string str;
            cin >> str;
            check(str);
        }
    }

    return 0;
}
```

**Функция check.**

Функция проверяет являются ли поданные данные синтаксически корректными. Явным признаком успешной проверки (в случае если выражение является логическим выражением) служит то, что должно выполниться условия положительных ответов из рекурсии, и восстановленная по иерархическому списку строка должна совпасть с исходной.

```
int check(string name) {

    List *list = new List;
    list->head = new Node;
    list->head->isAtom = true;
    list->head->next = nullptr;


    int c = 0;

    if(isLogicExpression(name, &c, list->head)){
        if("("+getString(list)+")"==name){
            cout << "OK"<<"\n";
        } else{
            cout << "NOT OK"<<"\n";
        }
    } else{
        cout << "NOT OK"<<"\n";
    }

    deleteAll(list);
    delete (list->head);
    delete (list);
```

```
    return 0;
}
```

### **3.3** Вспомогательные Функции

*1) isAnd (string basicString, int count)*

*2) isNot (string basicString, int count)*

*3) isOrImplication (string basicString, int count)*

*4) string getString(List *list)*

*5) deleteAll(List *list)*

```
bool isOrImplication(string myString, int num) {
    if (myString[num] == 'O' && myString[num + 1] == 'R' || myString[num] == '-' &&
myString[num + 1] == '>') {
        return true;
    }
    return false;
}

bool isAnd(string myString, int num) {
    if (myString[num] == 'A' && myString[num + 1] == 'N' && myString[num + 2] == 'D') {
        return true;
    }
    return false;
}

bool isNot(string myString, int num) {
    if (myString[num] == 'N' && myString[num + 1] == 'O' && myString[num + 2] == 'T') {
        return true;
    }
    return false;
}

}
```

Функции проверяют принадлежность кусочка строки (на который указывает параметр int count) к словам OR, AND, NOT и -> (знаку импликации).

Функция *getString(List *list) обходит иерархический список и собирает строку для проверки с исходной строкой.*

```
string getString(List *list) {

    string my_string = "";

    if (list->head != nullptr) {
        Node *my_node = list->head;
        while (my_node->next != nullptr) {
            if (my_node->isAtom) {
                my_string += my_node->my_data;
                my_node = my_node->next;
            } else {
                my_string += "(";
                my_string += getString(my_node->list);
                my_string += ")";
                my_node = my_node->next;
            }
        }
        if (my_node->isAtom) {
            string abc = my_node->my_data;
            my_string += abc;
        } else {
            my_string += "(";
            my_string += getString(my_node->list);
            my_string += ")";
        }
    }

    return my_string;
}
```

Функция *deleteAll(Lisst *list) обходит иерархический список и освобождает память.*

```
void deleteAll(List *list) {

    if (list->head != nullptr) {
        Node *my_node = list->head;
        while (my_node->next != nullptr) {
            if (my_node->isAtom) {
                Node *tmp = my_node->next;
                delete(my_node);
                my_node = tmp;
            } else {
                getString(my_node->list);
                Node *tmp = my_node->next;
                delete(my_node);
                my_node = tmp;
            }
```

```
        }
        if (my_node->isAtom) {
            delete(my_node);
        } else {
            getString(my_node->list);
        }
    }

}
```

**3.4** Функция *bool isLogicExpression(string myString, int *count, Node *node)* проверяет кусочек строки на принадлежность к понтию выражение. Также функция добавляет значение в элемент головы списка.

```
bool isLogicExpression(string myString, int *count, Node *node) {
    int num = *count;
    if (myString[num] == '(') {
        if (isOrImplication(myString, num + 1)) {
            *count = *count + 3;
            if(myString[num+1]=='O') {
                node->my_data = "OR";
            } else{
                node->my_data = "->";
            }
            if (isEquenceArguments(myString, count, node)) {
                int s = *count;
                if (myString[s] == ')') {
                    *count = *count + 1;
                    return true;
                }
            }
        }
        if (isAnd(myString, num + 1)) {
            *count = *count + 4;
            node->my_data = "AND";
            if (isEquenceArguments(myString, count, node)) {
                int s = *count;
                if (myString[s] == ')') {
                    *count = *count + 1;
                    return true;
                }
            }
        }
    }
    return false;
}
```

**3.5** Функция *bool isEquenceArguments(string myString, int *count, Node *node)* проверяет кусочек строки на принадлежность к набору аргументов. Набор аргументов – это 1 и более аргументов, находящихся друг за другом.

```
bool isEquenceArguments(string myString, int *count, Node *node) {
    int countArguments = 0;
    Node *tmp = node;
    while (isArgument(myString, count, tmp)) {
        tmp = tmp->next;
        countArguments++;
    }
    return countArguments > 0;
}
```

**3.6** Функция *bool isArgument(string myString, int *count, Node *node)* проверяет кусочек строки на принадлежность значения к понятию аргумента. В случае принадлежности добавляется новый элемент, являющийся атомом, в противном случае добавляется новый элемент, не являющийся атомом и ссылающийся на список.

```
bool isArgument(string myString, int *count, Node *node) {
    int num = *count;
    if (isalpha(myString[num]) && islower(myString[num])) {
        *count = *count + 1;
        string aaa = {myString[num]};

        Node *my_node = node;
        Node *newNode = new Node;
        newNode->my_data = aaa;
        newNode->isAtom = true;
        newNode->next = nullptr;
        my_node->next = newNode;

        return true;
    }
    if (myString[num] == '(' && isNot(myString, num + 1)) {

        if (isalpha(myString[num + 4]) && islower(myString[num + 4])) {

            if (myString[num + 5] == ')') {
                *count = *count + 6;
                string aaa = {myString[num], myString[num + 1], myString[num + 2], myS-
tring[num + 3], myString[num + 4],
                              myString[num + 5]};
```

```
                Node *my_node = node;
                Node *newNode = new Node;
                newNode->my_data = aaa;
                newNode->isAtom = true;
                newNode->next = nullptr;
                my_node->next = newNode;

                return true;
            }

        }

    }

    Node *newNode = new Node;
    newNode->isAtom = false;
    newNode->my_data = '#';
    newNode->list = new List;
    newNode->list->head = new Node;
    newNode->list->head->next = nullptr;
    newNode->list->head->isAtom = true;
    newNode->next = nullptr;

    if (isLogicExpression(myString, count, newNode->list->head)) {
        Node *my_node = node;
        my_node->next = newNode;
        return true;
    }
    return false;
}
```

# **4** Тесты

## 1.1 Покажем что, программа корректно работает на простых примерах :

Ответ ok

1) (ORq)

2) (ANDq)

3) (->qas)

4)

Ответ not ok

1) (OR)

2) (->)

3) AAAA

4) (ANq)

## 1.2 Покажем что, программа корректно работает на простых рекурсивных примерах (операция, операнды, операнд):

Ответ ok

1) (->qNOT(a)s)

2) (AND(NOTc)a)

3) (ANDa(NOTb)c)

4) AND(TRUE)

Ответ not ok

1) (ANq)

2) (ANq.

3) ANq)

4) (->24)

1.3 Покажем что, программа корректно работает на сложных рекурсивных  примерах (операция, операнды, операнд):

Ответ ok

1) (AND(NOTq)qwer)

2) (->(NOTq)qwer(->aaa))

3) (ORasd(ORasdf(ORsdfg(ORdsfsdvdf))))

4) (AND(NOTq)w(ANDc)(AND(NOTq)w(ANDc)))

**Вывод:** *были построены иерархические списки для проверки синтаксической корректности выражения. Изучена работа рекурсии для задания иерархических списков. Были написаны тесты и проверена работоспособность программы.*

## Приложение

## Код программы lab2.cpp

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;
typedef struct List;
typedef struct Node;

typedef struct Node {
    string my_data;
    bool isAtom;
    Node *next;
    List *list;
};

typedef struct List {
    Node *head;
};
```

```
bool isOrImplication(string myString, int num);

bool isAnd(string myString, int count);

bool isNot(string myString, int count);

bool isArgument(string myString, int *count, Node *node);

bool isEquenceArguments(string myString, int *count, Node *node);

bool isLogicExpression(string myString, int *count, Node *node);

bool isOrImplication(string myString, int num) {
    if (myString[num] == 'O' && myString[num + 1] == 'R' || myString[num] == '-' &&
myString[num + 1] == '>') {
        return true;
    }
    return false;
}

bool isAnd(string myString, int num) {
    if (myString[num] == 'A' && myString[num + 1] == 'N' && myString[num + 2] == 'D') {
        return true;
    }
    return false;
}

bool isNot(string myString, int num) {
    if (myString[num] == 'N' && myString[num + 1] == 'O' && myString[num + 2] == 'T') {
        return true;
    }
    return false;
}

bool isArgument(string myString, int *count, Node *node) {
    int num = *count;
    if (isalpha(myString[num]) && islower(myString[num])) {
        *count = *count + 1;
        string aaa = {myString[num]};

        Node *my_node = node;
        Node *newNode = new Node;
        newNode->my_data = aaa;
        newNode->isAtom = true;
        newNode->next = nullptr;
        my_node->next = newNode;

        return true;
    }
    if (myString[num] == '(' && isNot(myString, num + 1)) {

        if (isalpha(myString[num + 4]) && islower(myString[num + 4])) {

            if (myString[num + 5] == ')') {
                *count = *count + 6;
                string aaa = {myString[num], myString[num + 1], myString[num + 2], myS-
tring[num + 3], myString[num + 4],
                              myString[num + 5]};

                Node *my_node = node;
```

```
                Node *newNode = new Node;
                newNode->my_data = aaa;
                newNode->isAtom = true;
                newNode->next = nullptr;
                my_node->next = newNode;

                return true;
            }

        }

    }

    Node *newNode = new Node;
    newNode->isAtom = false;
    newNode->my_data = '#';
    newNode->list = new List;
    newNode->list->head = new Node;
    newNode->list->head->next = nullptr;
    newNode->list->head->isAtom = true;
    newNode->next = nullptr;

    if (isLogicExpression(myString, count, newNode->list->head)) {
        Node *my_node = node;
        my_node->next = newNode;
        return true;
    }
    return false;
}

bool isEquenceArguments(string myString, int *count, Node *node) {
    int countArguments = 0;
    Node *tmp = node;
    while (isArgument(myString, count, tmp)) {
        tmp = tmp->next;
        countArguments++;
    }
    return countArguments > 0;
}

bool isLogicExpression(string myString, int *count, Node *node) {
    int num = *count;
    if (myString[num] == '(') {
        if (isOrImplication(myString, num + 1)) {
            *count = *count + 3;
            if(myString[num+1]=='O') {
                node->my_data = "OR";
            } else{
                node->my_data = "->";
            }
            if (isEquenceArguments(myString, count, node)) {
                int s = *count;
                if (myString[s] == ')') {
                    *count = *count + 1;
                    return true;
                }
            }
        }
        if (isAnd(myString, num + 1)) {
            *count = *count + 4;
            node->my_data = "AND";
```

```
            if (isEquenceArguments(myString, count, node)) {
                int s = *count;
                if (myString[s] == ')') {
                    *count = *count + 1;
                    return true;
                }
            }
        }
    }
    return false;
}

string getString(List *list) {

    string my_string = "";

    if (list->head != nullptr) {
        Node *my_node = list->head;
        while (my_node->next != nullptr) {
            if (my_node->isAtom) {
                my_string += my_node->my_data;
                my_node = my_node->next;
            } else {
                my_string += "(";
                my_string += getString(my_node->list);
                my_string += ")";
                my_node = my_node->next;
            }
        }
        if (my_node->isAtom) {
            string abc = my_node->my_data;
            my_string += abc;
        } else {
            my_string += "(";
            my_string += getString(my_node->list);
            my_string += ")";
        }
    }

    return my_string;
}

void deleteAll(List *list) {

    if (list->head != nullptr) {
        Node *my_node = list->head;
        while (my_node->next != nullptr) {
            if (my_node->isAtom) {
                Node *tmp = my_node->next;
                delete(my_node);
                my_node = tmp;
            } else {
                getString(my_node->list);
                Node *tmp = my_node->next;
                delete(my_node);
                my_node = tmp;
            }
        }
        if (my_node->isAtom) {
            delete(my_node);
        } else {
```

```
            getString(my_node->list);
        }
    }

}

int check(string name) {

    List *list = new List;
    list->head = new Node;
    list->head->isAtom = true;
    list->head->next = nullptr;


    int c = 0;

    if(isLogicExpression(name, &c, list->head)){
        if("("+getString(list)+")"==name){
            cout << "OK"<<"\n";
        } else{
            cout << "NOT OK"<<"\n";
        }
    } else{
        cout << "NOT OK"<<"\n";
    }

    deleteAll(list);
    delete (list->head);
    delete (list);

    return 0;
}


int main() {

    int your_choose = 0;

    cout << "If you want to enter data from a file, enter \'1\'\n";
    cout << "If you want to enter data manually, enter \'2\'\n";

    cin >> your_choose;

    if (your_choose==1) {
        ifstream fin;
        fin.open("test2.txt");

        if (fin.is_open()) {
            cout << "Reading from file:" << "\n";

            int super_count = 0;

            while (!fin.eof()) {

                super_count++;

                string str;
                getline(fin, str);

                cout << "test #" << super_count << " \"" + str + "\"" << "\n";
                check(str);
```

```
            }
        } else {
            cout << "File not opened";
        }

        fin.close();
    } else{
        if(your_choose==2){
            cout << "Enter data \n";
            string str;
            cin >> str;
            check(str);
        }
    }

    return 0;
}
```