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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Тема: Рекурсивная обработка иерархических списков

Студент(ка) гр. 9382 _____ Русинов Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Научиться рекурсивно работать с иерархическими списками.

Задание.

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

Основные теоретические положения.

В практических приложениях возникает необходимость работы с более сложными, чем линейные списки, нелинейными конструкциями. Рассмотрим иерархичесикй список элементов базового типа El или S-выражение.

Определим соответствующий тип данных S_expr (El) рекурсивно,

используя определение линейного списка (типа L_list):

< S_expr (El) > ::= < Atomic (El) > | < L_list (S_expr (El)) >,

< Atomic (E) > ::= < El >.

< L_list(El) > ::= < Null_list > | < Non_null_list(El) >

< Null_lis t> ::= Nil

< Non_null_list(El) > ::= < Pair(El) >

< Pair(El) > ::= (< Head_l(El) > . < Tail_l(El) >)

< Head_l(El) > ::= < El >

< Tail_l(El) > ::= < L_list(El) >

Струтура иерархического списка:

```
typedef char base; // базовый тип элементов (атомов)
        struct s_expr;
        struct two_ptr {
        s_expr *hd;
        s_expr *tl;
} ; //end two_ptr;
struct s_expr {
```

```
        bool tag; // true: atom, false: pair
        union {
                base atom;
                two_ptr pair;
        } node;
                //end union node
};
//end s_expr
typedef s_expr *lisp;
```

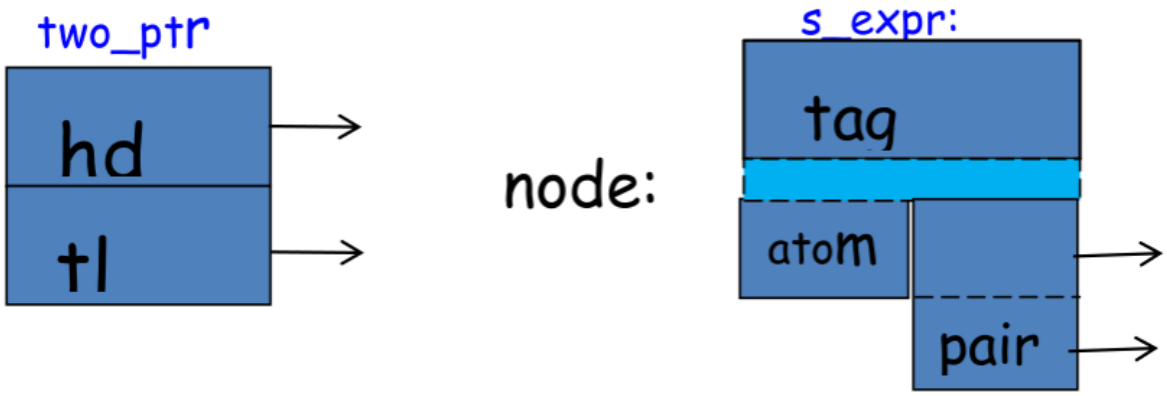
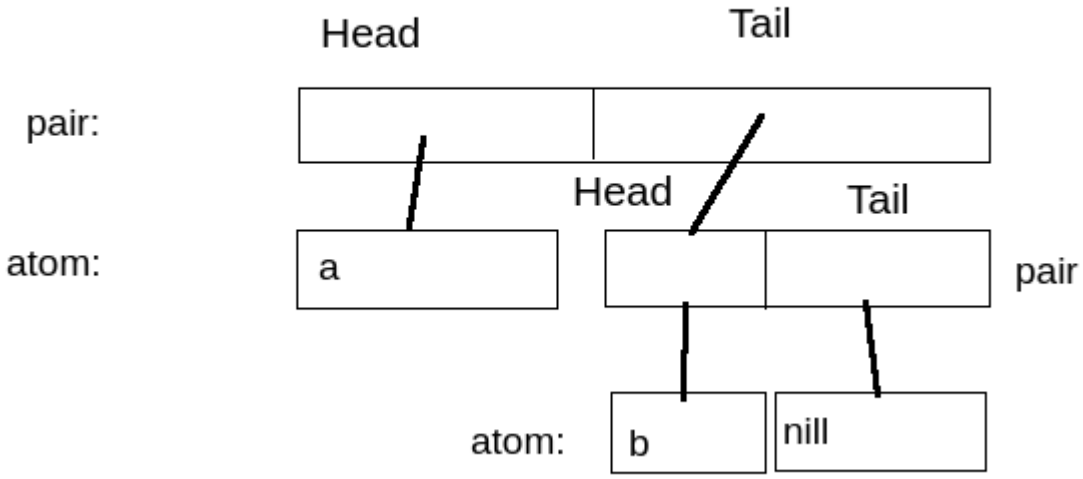


Рис. 2.3. Представление рекурсивной структуры списка

(a (b))



Реализованные функции.

Реализован класс List – список.

Класс List имеет следующие поля:

1) List* next – следующий элемент списка
2) List* child – предок элемента списка
3) char value – значение элемента списка, в случае если элемент является атомом

У класса List реализован конструктор и деструктор, а также следующие методы:

1) void addNextElement(List* element) – добавить элемент в список
2) void addChild(List* element) – добавить предка к элементу списка.
3) bool isAtom() const – является ли элемент списка атомом
4) void viewList(std::string format=std::string()) – показать список
5) char getAtom() const – получить значение атома.

Реализован класс Expression – выражение.

Класс Expression имеет следующие поля:

1) std::string expression – строка с выражением
2) bool correctness – переменная, которая говорит, корректное ли выражение

Конструктор принимает на вход std::string – строку с выражением.

У класса Expression реализованы следующие методы:

1) void checkExpression(int* index) – метод проверяет выражение на синтаксическую корректность. Принимает int* index – индекс, с которого стоит начать проверку выражения.
2) char findOperator(const int* index) – метод, который ищет в выражении один из следующих операторов:
   - AND – (верн. ‘&’, принимает два операнда)
   - OR – (верн. ‘|’, принимает два операнда)
   - NAND – (верн. ‘~’, принимает два операнда)
   - NOT – (верн. ‘!’, принимает один операнд)

   Если ни один из этих операторов не найден, то возвращается 0.
3) int setOffset(char op, int* index) – метод, который устанавливает смещение текущего индекса проверяемого символа в выражении после найденного оператора. Например, если найден оператор AND, нужно установить смещение +3. Также метод возвращает кол-во операндов, который принимает оператор. Например, для AND вернется значение 2.
4) List* createList(int* index) – создает иерархический список из заданного выражения. Передается int* index – индекс в выражении, с которого следует начать создание иерархического списка.
5) void operator *() – переопределенный оператор *. Переопределен для удобства. Сначала создается выражение, конструктор вызывает метод проверки выражения. При вызове этого метода,

создается иерархический список из выражения, если в поле correctness хранится значение true, затем у списка вызывается метод viewList.

Описание рекурсивной функции.

В конструкторе Expression вызывается рекурсивный метод checkExpression, который проверяет заданное выражение. Он реализован по принципу синтаксического анализатора из первой лабораторной работы. Если встречена ‘(’, то встречено начало операции. В операции необходимо найти три компонента – оператор (AND, NAND, OR, NOT) и два операнда. В качестве операнда может быть операция или один из символов [a..z]. После найденного оператора устанавливается смещение проверяемого индекса и проверяется кол-во операндов, которое присуще данному оператору.

Метод createList реализован так же по принципу синтаксического анализатора, только без лишних проверок, поскольку все проверки были выполнены методом checkExpression. Каждый вызов createList возвращает список из текущей операции. Если вызов createList был вызван этой же функцией, и список является операцией (т.е. есть оператор AND, NAND, OR, NOT), то возвращаемый список будет записан в child элемента списка. Иначе возвращаемый список будет из одного элемента – [a..z], поэтому он добавляется через метод addNextElement.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР----------	Проверка на некорректных данных

		Длина выражения 1 и этот символ [a..z]. Выражение верное ---------- ИЕРАРХИЧЕСКИЙ СПИСОК---------- 1 - a ----------	
2.	(AND a)	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Под индексом - 0 найдено начало операции Найден оператор - & Количество операндов для оператора - 2 Проверка операнда Под индексом - 5 найден символ a Проверка операнда Не было встречено символа [a..z] или символа начала операции ---------- Дорогой друг, вероятно, в вашем выражении ошибка!	Проверка на некорректных данных
3.	()	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР----------	Проверка на некорректных данных

		Под индексом - 0 найдено начало операции В операции не найден оператор! ---------- Дорогой друг, вероятно, в вашем выражении ошибка!	
4.	(AND a b)	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Под индексом - 0 найдено начало операции Найден оператор - & Количество операндов для оператора - 2 Проверка операнда Под индексом - 5 найден символ a Проверка операнда Под индексом - 7 найден символ b Операция синтаксически корректная ---------- ИЕРАРХИЧЕСКИЙ СПИСОК---------- 1 - & 2 - a 3 - b	

5.	(AND a (NOT (NAND (OR a b) c)) c)	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Под индексом - 0 найдено начало операции Найден оператор - & Количество операндов для оператора - 2 Проверка операнда Под индексом - 5 найден символ a Проверка операнда Под индексом - 7 найдено начало операции Найден оператор - ! Количество операндов для оператора - 1 Проверка операнда Под индексом - 12 найдено начало операции Найден оператор - ~ Количество операндов для оператора - 2 Проверка операнда Под индексом - 18 найдено начало операции Найден оператор - |	Проверка на некорректных данных

		Количество операндов для оператора - 2 Проверка операнда Под индексом - 22 найден символ a Проверка операнда Под индексом - 24 найден символ b Операция синтаксически корректная Проверка операнда Под индексом - 27 найден символ c Операция синтаксически корректная Операция синтаксически корректная Последний символ операции не закрывающая скобка ---------- Дорогой друг, вероятно, в вашем выражении ошибка!	
6.	(AND a (NOT (NAND (OR a b) c)))	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Под индексом - 0 найдено начало операции Найден оператор - &	

		Количество операндов для оператора - 2 Проверка операнда Под индексом - 5 найден символ a Проверка операнда Под индексом - 7 найдено начало операции Найден оператор - ! Количество операндов для оператора - 1 Проверка операнда Под индексом - 12 найдено начало операции Найден оператор - ~ Количество операндов для оператора - 2 Проверка операнда Под индексом - 18 найдено начало операции Найден оператор - | Количество операндов для оператора - 2 Проверка операнда Под индексом - 22 найден символ a Проверка операнда Под индексом - 24 найден символ b	

		Операция синтаксически корректная Проверка операнда Под индексом - 27 найден символ c Операция синтаксически корректная Операция синтаксически корректная Операция синтаксически корректная ---------- ИЕРАРХИЧЕСКИЙ СПИСОК---------- 1 - & 2 - a 3.1 - ! 3.2.1 - ~ 3.2.2.1 - | 3.2.2.2 - a 3.2.2.3 - b 3.2.3 - c ----------	
7.	(AND a)	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Под индексом - 0 найдено начало операции Найден оператор - & Количество операндов для оператора - 2	Проверка на некорректных данных

		Проверка операнда Под индексом - 5 найден символ a После оператора/первого операнда не был встречен пробел ---------- Дорогой друг, вероятно, в вашем выражении ошибка!	
8.	(AND (NOT a) b)	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Под индексом - 0 найдено начало операции Найден оператор - & Количество операндов для оператора - 2 Проверка операнда Под индексом - 5 найдено начало операции Найден оператор - ! Количество операндов для оператора - 1 Проверка операнда Под индексом - 10 найден символ a Операция синтаксически корректная	

		Проверка операнда Под индексом - 13 найден символ b Операция синтаксически корректная ---------- ИЕРАРХИЧЕСКИЙ СПИСОК---------- 1 - & 2.1 - ! 2.2 - a 3 - b ----------	
9.	(NOT c)	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Под индексом - 0 найдено начало операции Найден оператор - ! Количество операндов для оператора - 1 Проверка операнда Под индексом - 5 найден символ c Операция синтаксически корректная ---------- ИЕРАРХИЧЕСКИЙ СПИСОК---------- 1 - ! 2 - c	

10.	NOT c	---------- СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР---------- Не было встречено символа [a..z] или символа начала операции ---------- Дорогой друг, вероятно, в вашем выражении ошибка!	Проверка на некорректных данных

Выводы.

Освоена работа с иерархическими списками, а также полученные знания были применены на практике.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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


class List {
    List* next = nullptr;
    List* child = nullptr;
    char value = 0;

public:
    List() = default;
    ~List();
    explicit List(char atom);
    void addNextElement(List* element);
    void addChild(List* element);
    bool isAtom() const;
    void viewList(std::string format = std::string());
    char getAtom() const;
};


class Expression {
    std::string expression;
    // Метод для проверки корректности выражения
    void checkExpression(int* index);
    // Метод для поиска оператора в выражении
    char findOperator(const int* index);
    // Метод для установки смещения в выражении после оператора
    // Возвращает количество аргументов, которое приинмает оператор
    int setOffset(char op, int* index);
public:
    bool correctness = true;
    explicit Expression(std::string string);
    // Метод для создания иерархического списка из выражения
    List* createList(int* index);
    // Переопределенный оператор, ну, просто для красоты
    void operator *();
};


// добавление в список след элемента
void List::addNextElement(List *element) {
    List* current = this;
    while (current->next) current = current->next;
    current->next = element;
}

// добавление в список child
void List::addChild(List *element) {
    child = element;
}
```

```
// проверка является ли атомом элемент
bool List::isAtom() const {
    if (value) return true;
    return false;
}

// конструктор
List::List(char atom) : value(atom) {}

List::~List() {
    delete next;
    delete child;
}

// визуализация списка
void List::viewList(std::string format) {
    List* current = this;
    int counter = 1;
    // пока есть элементы, будем их отображать
    while(current) {
        // если элемент не атом, значит у него есть child
        // отобразим child
        if (!current->isAtom()) {
            if (format.length()) current->child->viewList(format + '.' +
std::to_string(counter));
            else current->child->viewList(std::to_string(counter));
        } // если атом, то отобразим его значение
        else {
            if (format.length()) std::cout << format + "." +
std::to_string(counter) << " - " << current->value << std::endl;
            else std::cout << counter << " - " << current->value <<
std::endl;
        }
        counter++;
        current = current->next;
    }
}

char List::getAtom() const {
    return value;
}


Expression::Expression(std::string string) : expression(string) {
    // Если длина выражения равна нулю, то оно заведомо неверное
    std::cout << "----------СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР----------" <<
std::endl;
    if (expression.length() == 0) {
        std::cout << "Длина выражения меньше единицы!" << std::endl;
        correctness = false;
        return;
    }

    // Если длина выражения равна единице, и этот символ в [a..z], то вы-
ражение верное и не нуждается
    // в дополнительной проверке
```

```
    if ((97 <= expression[0]) && (expression[0] <= 122) && expres-
sion.length() == 1) {
        std::cout << "Длина выражения 1 и этот символ [a..z]. Выражение
верное" << std::endl;
        return;
    }

    // Если длина выражения больше единицы, и первый символ [a..z], то
выражение заведомо неверное
    if ((97 <= expression[0]) && (expression[0] <= 122) && expres-
sion.length() > 1) {
        std::cout << "Длина выражения > 1, и первый символ [a..z], выра-
жение неверное" << std::endl;
        correctness = false;
        return;
    }

    // Основная синтаксическая проверка выражения находится здесь
    int index = 0;
    checkExpression(&index);


    // Если выражение было проверено и оно все еще корректное
    // Проверим, что индекс находится на конце выражения
    if (correctness) {
        if (index != expression.length()) {
            correctness = false;
            std::cout << "Не удалось дойти до конца выражения! Есть лиш-
ние символы" << std::endl;
        }
    }
}

void Expression::checkExpression(int* index) {
    // Если мы вышли за границы выражения, то оно неверное
    if (*index >= expression.length()) {
        std::cout << "Текущий индекс - " << *index << std::endl;
        std::cout << "Длина выражения - " << expression.length() << ".
Выход за границу" << std::endl;
        correctness = false;
        return;
    }

    // Если символ под индексом [a..z], то выражение верное
    if ((97 <= expression[*index]) && (expression[*index] <= 122)) {
        std::cout << "Под индексом - " << *index << " найден символ " <<
expression[*index] << std::endl;
        *index += 1;
    }
        // Если встречен символ '(', то была встречена операция
    else if (expression[*index] == '(') {
        std::cout << "Под индексом - " << *index << " найдено начало опе-
рации" << std::endl;
        *index += 1;
        // Пытаемся найти оператор
        char op = findOperator(index);
        // Если не нашелся оператор, то выражение неверное
        if (!op) {
```

```
            std::cout << "В операции не найден оператор!" << std::endl;
            correctness = false;
            return;
        }

        std::cout << "Найден оператор - " << op << std::endl;

        // Устанавливаем смещение индекса и получаем количество операндов
для оператора
        int countOperands = setOffset(op, index);
        std::cout << "Количество операндов для оператора - " << countOp-
erands << std::endl;

        // Если вышли за границы выражения после установки смещения,
        // то выражение неверное
        if (*index >= expression.length()) {
            std::cout << "Текущий индекс - " << *index << std::endl;
            std::cout << "Длина выражения - " << expression.length() <<
". Выход за границу" << std::endl;
            correctness = false;
            return;
        }

        // Циклически проверяем операнды операции
        for (int i = 0; i < countOperands; ++i){
            // Если встречен пробел, значит все впорядке и можно прове-
рять операнд
            if (expression[*index] == ' ') {
                *index += 1;
                std::cout << "Проверка операнда" << std::endl;
                checkExpression(index);
                if (!correctness) return;
                // если после проверки операнда вышли за границы выраже-
ния
                // то выражение некорректно
                if (*index >= expression.length()) {
                    std::cout << "Текущий индекс - " << *index <<
std::endl;
                    std::cout << "Длина выражения - " << expres-
sion.length() << ". Выход за границу" << std::endl;
                    correctness = false;
                    return;
                }
                // пробел не встречен, значит выражение некорректное
            } else {
                std::cout << "После оператора/первого операнда не был
встречен пробел" << std::endl;
                correctness = false;
                return;
            }
        }

        // если последний символ операции не закрывающая скобка
        // значит выражение неверное
        if (expression[*index] != ')') {
            std::cout << "Последний символ операции не закрывающая
скобка" << std::endl;
            correctness = false;
```

```
            return;
        } else {
            *index += 1;
            std::cout << "Операция синтаксически корректная" <<
std::endl;
        }

    }
        // если был встречен символ не [a..z] и не '(', то выражение не-
верное
    else {
        std::cout << "Не было встречено символа [a..z] или символа начала
операции" << std::endl;
        correctness = false;
        return;
    }
}

char Expression::findOperator(const int* index) {
    // ищем оператор NOT
    int found = expression.find("NOT", *index);
    if (found != std::string::npos && found == *index) return '!';

    // ищем оператор AND
    found = expression.find("AND", *index);
    if (found != std::string::npos && found == *index) return '&';

    // ищем оператор OR
    found = expression.find("OR", *index);
    if (found != std::string::npos && found == *index) return '|';

    // ищем оператор NAND
    found = expression.find("NAND", *index);
    if (found != std::string::npos && found == *index) return '~';

    // операторы не найдены, возвращаем 0
    return 0;
}

int Expression::setOffset(char op, int* index) {
    int countOperands;
    // если оператор NOT или AND
    // то смещение - 3 символа
    // количество аргументов 1, если NOT
    // 2, если AND
    if (op == '!' || op == '&') {
        *index += 3;
        if (op == '!') countOperands = 1;
        else countOperands = 2;
    }
        // если оператор OR, смещение - 2, кол-во аргументов - 2
    else if (op == '|') {
        *index += 2;
        countOperands = 2;
    }
        // в другом случае NAND, смещение - 4, кол-во аргументов - 2
    else {
        *index += 4;
```

```
        countOperands = 2;
    }
    return countOperands;
}

List* Expression::createList(int* index) {
    // создание списка уже заведомо верного выражения
    // поэтому можно избежать различных проверок на корректность выраже-
ния

    if ((97 <= expression[*index]) && (expression[*index] <= 122)) {
        // если символ [a..z], то создаем из него элемент списка и воз-
вращаем
        List* list = new List(expression[*index]);
        *index += 1;
        return list;
    } else {
        // тогда символ '(' и это операция
        *index += 1;

        // находим оператор
        char op = findOperator(index);
        // создаем элемент списка из оператора
        List* list = new List(op);
        // кол-во аргументов для оператора и смещение
        int countOperands = setOffset(op, index);

        // считывание операндов в цикле
        for (int i = 0; i < countOperands; ++i){
            *index += 1;
            // создаем элемент списка из операнда
            List* operand = createList(index);
            // если оказалось, что первым элементом списка операнда явля-
ется оператор
            // значит это child
            if (operand->getAtom() == '!' ||
                operand->getAtom() == '~' ||
                operand->getAtom() == '&' ||
                operand->getAtom() == '|') {
                // создаем новый элемент списка
                List* nextEl = new List();
                // добавляем к нему child
                nextEl->addChild(operand);
                // к списку с оператором добавляем элемент с child
                list->addNextElement(nextEl);
            } // иначе это просто следующий элемент
            else list->addNextElement(operand);
        }
        *index += 1;
        return list;
    }
}

void Expression::operator*() {
    // оператор вывода результата
    if (!correctness) {
        std::cout << "-----------------------------------------" <<
std::endl;
```

```
        std::cout << "Дорогой друг, вероятно, в вашем выражении ошибка!";
    } else {
        int index = 0;
        List* list = createList(&index);
        std::cout << "------------ИЕРАРХИЧЕСКИЙ СПИСОК-------------" <<
std::endl;
        list->viewList();
        std::cout << "---------------------------------------------" <<
std::endl;
        delete list;
    }
}


int main() {
    std::string string;

    std::ifstream in("../file.txt");
    if (!in.is_open()) {
        std::cout << "Не удалось открыть файл!" << std::endl;
        in.close();
        return 0;
    }

    std::getline(in, string, '\n');
    in.close();

    *Expression(string);
    return 0;
}
```