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: Иерархические списки

Студент(ка) гр. 9381 _____ Шахин Н.С

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с основными понятиями и приёмами рекурсивной обработки списков, изучить особенности реализации иерархического списка на языке программирования C++. Разработать программу, использующую иерархические списки и их рекурсивную обработку, анализирующую корректность выражения.

Задание.

Вариант 22.

Пусть алгебраическое выражение представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в постфиксной форме ( )). Аргументов может быть 1, 2 и более. Например (в префиксной форме): (+ a (* b (! c))). Необходимо реализовать обработку алгебраических выражений в префиксной форме (+, -, *, sqrt(), log(,)), проверку синтаксической корректности, простую проверку log(,).

Основные теоретические положения.

Согласно рекурсивному определению, иерархический список – это список, элементами которого так же могут быть иерархические списки. Для обработки иерархического списка используются рекурсивные функции, так как он представляет собой множество списков, между которыми установлена иерархия.

На рисунке 1 представлен иерархический список, обрабатываемый созданной программой. Список соответствует сокращенной записи **((a b) c d)**.

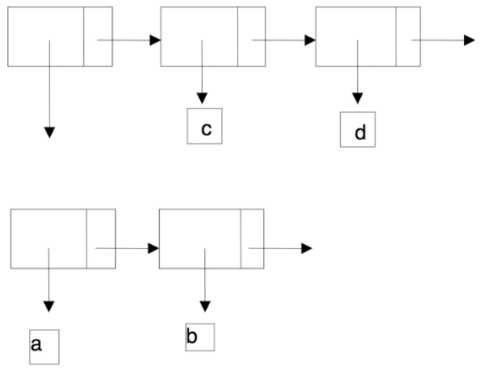

Рисунок 1. **((a b) c d)**.

Описание работы алгоритма.

На вход алгоритму передаётся строка, затем с помощью взаимно рекурсивных функций scanExpression(), scanTail(), scanNode() создаётся иерархический список. После создания каждого следующего Expression и добавления всех его атомов в список происходит проверка на корректность этого выражения. Алгоритм завершает работу, когда в строке не остаётся символов, или какое-либо Expression не соответствует условию.

Структуры для хранения Иерархического списка.

```
Листинг 1.
enum Type {    - виды выражений
    PLUS,
    MINUS,
    MULTIPLY,
    SQRT,
    LOG
};

struct Expression;

struct Node{         - элемент списка
    bool isNumber; true – число; false – указатель на выражение(список)
    union {
        int number; - число
        Expression* expression; - указатель на выражение(список)
    } data;
};

struct two_ptr{
    Node* hd;    - указатель на голову списка
    two_ptr* tl; - указатель на хвост списка
};

struct Expression{
    Type expressionType; тип выражения
    two_ptr* pair; указатель на голову и хвост списка
};
```

Функции для работы со списком и вывода результатов работы алгоритма.

void skip (string& str, int& pos, int n = 1, int indent = 0) - функция для удаления проверенных символов строки. В функцию передаются string& str - строка по ссылке, int& pos - текущее положение в строке, int indent - глубина

рекурсии, int n - количество символов для удаления. Перед удалением вызывается функция для вывода удаляемых символов.

Expression* scanExpression(string& input, int& pos, int indent) – функция для создания и проверки списка. В функцию передаются string& input – входная строка, int& pos – текущее положение в строке, int indent – глубина рекурсии. Функция возвращает указатель на список.

Node* scanNode(string& input, int& pos, int indent) – функция для создания головы списка. В функцию передаются string& input – входная строка, int& pos – текущее положение в строке, int indent – глубина рекурсии. Функция возвращает указатель на элемент списка.

two_ptr* scanTail(string& input, int& pos, bool isFunc, int indent) – функция для создания хвоста списка. В функцию передаются string& input – входная строка, int& pos – текущее положение в строке, int indent – глубина рекурсии, bool isFunc – является ли Expression операцией или функциями log, sqrt. От этой переменной зависит выбор разделителей при записи. Функция возвращает указатель на хвост списка.

int getNum(string& input, int& pos, int indent) – Функция для преобразования строки в число. В функцию передаются string& input – входная строка, int& pos – текущее положение в строке, int indent – глубина рекурсии. Функция возвращает число.

int length(two_ptr* list) – функция для подсчёта количества элементов в списке. В функцию передаётся указатель на хвост списка two_ptr* list. Функция возвращает количество элементов.

void logCheck(two_ptr* list, int& pos) – функция для проверки логарифма. В функцию передаётся указатель на хвост списка two_ptr* list, положение в строке int& pos для вывода ошибки. Если в списке не 2 элемента, то выводится ошибка. Если основание < 0 или равно 1 или показатель отрицательный, то выводится сообщение об ошибке.

void sqrtCheck(two_ptr* list, int& pos) – функция для проверки корня. В функцию передаётся указатель на хвост списка two_ptr* list, положение в

строке int& pos для вывода сообщения об ошибке. Если в списке не один элемент, то выводится сообщение об ошибке. Если в списке один элемент, но он отрицательный, то выводится сообщение об ошибке.

Взаимно рекурсивные функции для отчистки памяти:

void destroy(Expression*& lisp) отчищает Expression

void destroy(Node*& lisp) отчищает Node

void destroy(two_ptr*& lisp) отчищает two_ptr

Тестирование программы.

Для удобства тестирования программы был написан скрипт, вызывающий программу со всеми тестами из папки tests одной командой. Тестирование проводилось под операционной системой linux.

Таблица 1. Результаты тестирования

№	Входные данные	Выходные данные
1	(+ 2 3 (- 6 7 (* 2 log(2,4))))	START FUNCTION SCAN_EXPRESSION (+ START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE 2 END FUNCTION SCAN_NODE START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE 3 END FUNCTION SCAN_NODE START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE START FUNCTION SCAN_EXPRESSION (- START FUNCTION SCAN_TAIL

		START FUNCTION SCAN_NODE 6 END FUNCTION SCAN_NODE START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE 7 END FUNCTION SCAN_NODE START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE START FUNCTION SCAN_EXPRESSION (* START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE 2 END FUNCTION SCAN_NODE START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE START FUNCTION SCAN_EXPRESSION log (START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE 2

		END FUNCTION SCAN_NODE , START FUNCTION SCAN_TAIL START FUNCTION SCAN_NODE 4 END FUNCTION SCAN_NODE END FUNCTION SCAN_TAIL END FUNCTION SCAN_TAIL) END FUNCTION SCAN_EXPRESSION END FUNCTION SCAN_NODE END FUNCTION SCAN_TAIL END FUNCTION SCAN_TAIL) END FUNCTION SCAN_EXPRESSION END FUNCTION SCAN_NODE END FUNCTION SCAN_TAIL END FUNCTION SCAN_TAIL END FUNCTION SCAN_TAIL) END FUNCTION SCAN_EXPRESSION END FUNCTION SCAN_NODE END FUNCTION SCAN_TAIL

		END FUNCTION SCAN_TAIL END FUNCTION SCAN_TAIL) END FUNCTION SCAN_EXPRESSION (+ 2 3 (- 6 7 (* 2 log(2,4)))) SUCCESS
2	./myscript	test 1 read from file - tests/test.txt (+ 2 3 4 5) SUCCESS test 2 read from file - tests/test1.txt (+ 7 -8 9 10 sqrt(15) (- 4 5 6 -7) (- -1)) SUCCESS test 3 read from file - tests/test2.txt (+ log(3,5) (* 3 log(sqrt(16),2))) SUCCESS test 4 read from file - tests/test3.txt > 8: Log argument can't be negative sqrt(log(-3,2)) .....^ ERROR test 5 read from file - tests/test4.txt > 16: Sqrt must have 1 argument (+ (- (* log(sqrt(8,5),3)))) .....^ ERROR test 6 read from file - tests/test5.txt 15: End of expression expected (+ 8 7 (- 7 8)) 8

		.....^ ERROR test 7 read from file - tests/test6.txt > 22: Log base can't be negative or equals 1 (+ (- -7) 8 9 (* log(2,1))) .....^ ERROR test 8 read from file - tests/test7.txt (+ (- -7) 8 9 (* log(2,6))) SUCCESS test 9 read from file - tests/test8.txt > 4: Sqrt from negative number sqrt(-8) ....^ ERROR test 10 read from file - tests/test9.txt > 10: Unknown operator (+ 5 6 (-)) .....^ ERROR test 11 read from file - tests/test10.txt > 10: Unknown operator (+ 4 5 6 (())) .....^ ERROR test 12 read from file - tests/test11.txt (+ 2 (- 4 2) (* 2 3 4)) SUCCESS

		test 13 read from file - tests/test12.txt > 0: Unknown operator 1 2 3 4 5 ^ ERROR test 14 read from file - tests/test13.txt > 7: Unknown operator (+ 2 3 g) .....^ ERROR test 15 read from file - tests/test14.txt > 11: Expression close bracket not found (+ (- -1) 5 .....^ ERROR test 16 read from file - tests/test15.txt (* 3 4 5 log(2,3) (- -1 3) 3 (+ 4 5 (* sqrt(36)))) SUCCESS

Вывод.

Была создана программа, обрабатывающая алгебраическое выражение, хранящееся в иерархическом списке.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

файл structs.h

```
#ifndef AISD_LB2_STRUCTS_H
#define AISD_LB2_STRUCTS_H

#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>
#include <fstream>

using namespace std;

#endif //AISD_LB2_STRUCTS_H
```

файл main.cpp

```
#include <iostream>
#include "lisp.h"

int main (int argc, char** argv){
    string input = cuinput(argc, argv);
    if(input.empty()){
        cout<<"ERROR";
        return 0;
    }
    string copy = string(input);
    int pos = 0;
    errorFlag = false;
    int indent = 1;
    Expression * expression = scanExpression(copy, pos, indent);
    if (!copy.empty() && !errorFlag) {
        cout << pos << ": End of expression expected" << endl;
        errorPos = pos;
        errorFlag = true;
    }
    cout << input << endl;
    if (errorFlag) {
        for (int i = 0; i < errorPos; i++) {
            cout << '.';
        }
        cout << "^" << endl << "ERROR\n";
    }
    else {
        cout << "SUCCESS\n";
    }
    if(argc > 1){
        if(errorFlag) {
            ofstream outfile(optarg, ios::app);
            outfile << "ERROR\n";
        } else{
            ofstream outfile(optarg, ios::app);
            outfile << "SUCCESS\n";
```

```
        }
    }
    destroy(expression);
    return 0;
}
```

файл InOut.h

```
#ifndef AISD_LB2_INOUT_H
#define AISD_LB2_INOUT_H

#include "structs.h"
extern bool errorFlag;
extern int errorPos;

string cuinput(int argc, char** argv);

void proceedOutput(string output, int indent);

void proceedErr(const string& err, int& pos);

void info(int indent, int flag);

#endif //AISD_LB2_INOUT_H
```

файл InOut.cpp

```
#include "InOut.h"

bool errorFlag;
int errorPos;

string cuinput(int argc, char** argv){
    if(argc == 1) {
        cout<<"Write a statement: ";
        string res;
        getline(cin, res);
        return res;
    }
    int option = 0;
    while ((option = getopt(argc,argv,"hf:"))!=-1){
        switch (option) {
            case 'h': cout<<"If you want read from file use flaf -
f<filename>\n"; return "HELP";
            case 'f': cout<<"read from file - "<<optarg<<endl;
                ifstream infile(optarg);
```

```
                if (!infile) {
                    cout << "> File can't be open!" << endl;
                    return "";
                }
                string str;
                getline(infile, str);
                return str;
            }
        }
        return "";
}

void proceedOutput(string output, int indent){
    if(output != " ") {
        for (int i = 0; i < indent; i++) {
            cout << "  ";
        }
        cout << output << endl;
    }
}

void proceedErr(const string& err, int& pos){
    if (!errorFlag) {
        errorFlag = true;
        errorPos = pos;
        cout << "> " << pos << ": " << err << endl;
    }
}

void info(int indent, int flag){
    for(int i = 0; i < indent; i++){
        cout<<"  ";
    }
    if(flag == 1){
        cout <<"START FUNCTION SCAN_EXPRESSION\n";
    } else if(flag == 2){
        cout <<"END FUNCTION SCAN_EXPRESSION\n";
    }else if(flag == 3){
        cout <<"START FUNCTION SCAN_NODE\n";
    }else if(flag == 4){
        cout <<"END FUNCTION SCAN_NODE\n";
    }else if(flag == 5){
        cout <<"START FUNCTION SCAN_TAIL\n";
    }else if(flag == 6){
        cout <<"END FUNCTION SCAN_TAIL\n";
    }
}
```

файл lisp.h

```
#ifndef AISD_LB2_LISP_H
#define AISD_LB2_LISP_H

#include "InOut.h"

enum Type {
    PLUS,
    MINUS,
    MULTIPLY,
    SQRT,
```

```
    LOG
};

struct Expression;

struct Node{
    bool isNumber;
    union {
        int number;
        Expression* expression;
    } data;
};

struct two_ptr{
    Node* hd;
    two_ptr* tl;
};

struct Expression{
    Type expressionType;
    two_ptr* pair;
};

Expression* scanExpression(string& input, int& pos, int indent);
Node* scanNode(string& input, int& pos, int indent);
two_ptr* scanTail(string& input, int& pos, bool isFunc, int indent);
void skip (string& str, int& pos, int n = 1, int indent = 0);
int getNum(string& input, int& pos, int indent);
int length(two_ptr* list);

void logCheck(two_ptr* list, int& pos);
void sqrtCheck(two_ptr* list, int& pos);

void destroy(Expression*& lisp);
void destroy(Node*& lisp);
void destroy(two_ptr*& lisp);

#endif //AISD_LB2_LISP_H
```

файл lisp.cpp

```
#include "lisp.h"

void skip(string& str, int& pos, int n, int indent){
    if (str.length() >= n) {

        proceedOutput(str.substr(0, n), indent);

        str = str.substr(n);
        pos++;
    }
}

Expression* scanExpression(string& input, int& pos, int indent){

    info(indent, 1);

    if(input.empty()) return nullptr;
    bool isFunc = input[0] != '(';
    Expression* expression = new Expression;
    if(isFunc){
        if(input.find("log") == 0){
```

```
            expression->expressionType = LOG;
            skip(input, pos, 3, indent);
        }
        else if(input.find("sqrt") == 0){
            expression->expressionType = SQRT;
            skip(input, pos, 4, indent);
        }
        else {
            proceedErr("Unknown operator", pos);
        }
        if(input.empty() || input[0] != '('){
            proceedErr("Expected (", pos);
        }
        skip(input, pos, 1, indent);
        expression->pair = scanTail(input, pos, true, indent+1); // составляем
список из оставшеяся строки
        if(expression->pair == nullptr){
            proceedErr("Function arguments mot found", pos);
        }
        if(expression->expressionType == LOG){
            logCheck(expression->pair,   pos);   //   проверяем   корректонсть
аогументов логорифма
        }
        if(expression->expressionType == SQRT){
            sqrtCheck(expression->pair,   pos);   //   проверяем   коректность
аогументов sqrt
        }
        if(input.empty() || input[0] != ')'){ // проверяем закртую скобку после
аргументов логорифма
            proceedErr("Expected )", pos);
        }
        skip(input, pos, 1, indent);// убираем пробел
    }
    else{
        skip(input, pos, 1, indent); // удаляем (
        if(input.find('+') == 0){
            expression->expressionType = PLUS;
            skip(input, pos, 1, indent); // удаляем +
        } else if(input.find('-') == 0){
            expression->expressionType = MINUS;
            skip(input, pos, 1, indent); // удаляем -
        } else if(input.find('*') == 0){
            expression->expressionType = MULTIPLY;
            skip(input, pos, 1, indent); // удаляем *
        } else{
            proceedErr("Unknown operator", pos);
        }
        if (input.empty() || input[0] != ' '){
            proceedErr("Arguments not found", pos);
        }
        skip(input, pos, 1, indent); // удаляем пробел
        expression->pair = scanTail(input, pos, false, indent + 1);
        if(expression->pair == nullptr){
            proceedErr("Arguments not found", pos);
        }
        if (input.empty() || input[0] != ')')
            proceedErr("Expression close bracket not found", pos);
        skip(input, pos, 1, indent);// удаляем )
    }

    info(indent, 2);

    return expression;
```

```
}

two_ptr* scanTail(string& input, int& pos, bool isFunc, int indent){

    info(indent, 5);

    two_ptr* list = new two_ptr;
    list->hd = scanNode(input, pos, indent+1);
    if(list->hd == nullptr){ // если не записалась голова списка, то отчищаем
память и позвращаем nullprt
        delete list;

        info(indent, 6);

        return nullptr;
    }
    char sep = isFunc ? ',' : ' ';
    if(!input.empty() && input[0] == sep){
        skip(input, pos, 1, indent);
        list->tl = scanTail(input, pos, isFunc, indent+1);
    } else{
        list->tl = nullptr;
    }

    info(indent, 6);

    return list;
}

Node* scanNode(string& input, int& pos, int indent){

    info(indent, 3);

    if (input.empty() || input[0] == ')') {

        info(indent, 4);

        return nullptr;
    }
    Node* node  = new Node;
    node->isNumber = isdigit(input[0]) || (input.length() > 1 && input[0] == '-
' && isdigit(input[1]));
    if(node->isNumber){
        node->data.number = getNum(input, pos, indent);
    } else{
        node->data.expression = scanExpression(input, pos, indent+1);
    }

    info(indent, 4);

    return  node;
}

// преобразование строки в число
int getNum(string& str, int& pos, int indent){
    string strNum;
    while (isdigit(str[0]) || (strNum.length() == 0 && str[0] == '-')) {
        strNum += str[0];
        skip(str, pos, 1, indent);
    }
    return stoi(strNum);
}
```

```
// подсчёт количества атомов в списке
int length(two_ptr* list){
    two_ptr* l2 = list;
    if(l2 != nullptr && l2->hd != nullptr){
        int len = 0;
        while (l2 != nullptr){
            len++;
            l2 = l2->tl;
        }
        return len;
    } else return 0;
}

//Проверка аргументов логорифма
void logCheck(two_ptr* list, int& pos){
    if(length(list) == 2){ // если список состоит из двух атомов
        if(list->tl->hd->isNumber && (list->tl->hd->data.number == 1 || list-
>tl->hd->data.number < 0)){
            proceedErr("Log base can't be negative or equals 1", pos);
        }
        if(list->hd->isNumber && (list->hd->data.number < 0)){
            proceedErr("Log argument can't be negative", pos);
        }
    }else{
        proceedErr("Log must have 2 arguments", pos);
    }
}

// проверка аргумента корня
void sqrtCheck(two_ptr* list, int& pos){
    if(length(list) == 1){ // Если в списке один атом
        if (list->hd->isNumber && list->hd->data.number < 0) {
            proceedErr("Sqrt from negative number", pos);
        }
    }
    else {
        proceedErr("Sqrt must have 1 argument", pos);
    }
}

void destroy(Expression*& lisp){
    if(lisp != nullptr){
        destroy(lisp->pair);
        delete lisp;
    }
}

void destroy(Node*& lisp){
    if(lisp != nullptr){
        if(!lisp->isNumber){
            destroy(lisp->data.expression->pair);
            delete lisp;
        }
    }
}

void destroy(two_ptr*& lisp){
    if(lisp != nullptr){
        destroy(lisp->hd);
        destroy(lisp->tl);
        delete lisp;
    }
}
```