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 гр. 8381 _____ Сахаров В.М.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

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.

Пусть логическое выражение представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в постфиксной форме (<аргументы> <операция>)). 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 – такой список, элементами которого могут быть иерархические списки. В случае постфиксного выражения с помощью иерархического списка может быть построена зависимость операций (вложенность скобок).

Для обработки иерархического списка удобно использовать рекурсивные функции, так как он представляет собой множество линейный списков, между которыми установлены связи, иерархия.

Выполнение работы.

Написание работы производилось на базе операционной системы Windows 10 в среде разработки Jetbrains CLion с использованием фреймворка MinGW. Сборка, отладка и тестирование также производились в CLion.

Для реализации программы был разработан CLI с возможностью ввода с консоли, из файла и аргументов командной строки. Данные функции описаны в файле LabIo.h

Для выполнения задания была реализована структура Expression, которая, учитывая специфику входных данных, состоит из типа оператора и списка операндов. Структура для хранения операндов OperandList представляет из себя рекурсивный список, состоящий из указателя на головной элемент и на хвост списка. Элементы спика являются структурами Operand, в которых присутстует вариативность данных, выражающаяся в том, что они могут хранить как число, так и указатель на Expression, где и проявляется рекурсивность иерархического списка.

Функции ScanExpression, ScanOperandList, ScanOperand используются для парсинга входящей строки в соответственные структуры и принимают в качестве аргумента ссылку на входную строку и ссылку на абсолютный номер символа в строке для вывода в случае возникновения ошибки. В функции ScanOperandList также используется аргумент bool isClassic для возможности обработки как префиксной формы аргументов, так и классических функций log и sqrt.

Функции CheckLog и CheckSqrt проверяют соответственные списки на простейшие ошибки в количестве аргументов и их значений (например, основание логарифма не может быть отрицательным или равным 1)

Функция Length возвращает длину списка, проходя в цикле по всем его элементам до nullptr и считая их количество.

Функция GetNumber преобразует строковое представление числа в начале строки в число.

Функция Skip удаляет n символов из начала строки, используя метод substr

Функция ProceedError выводит первую встреченную ошибку и записывает это в глобальных переменных errorFlag и errorPos.

Оценка эффективности алгоритма.

Алгоритм, реализованный в программе, имеет линейную зависимость от длины строки, то есть сложность оценивается как O(n). Рассуждения отталкиваются от того, что внутри каждой функции-анализаторе будет удалён как минимум один символ на каждой итерации. Ввиду рекурсивного алгоритма рост

занимаемой памяти растет линейно из-за создаваемых в функциях временных переменных.

Тестирование программы.

Таблица 1: тестирование программы

Входные данные	Выходные данные
(+ 2 (- 4 2) (* 2 3 4))	(+ 2 (- 4 2) (* 2 3 4)) SUCCESS
(+ 8 7 (- 8 7)) 8	15: End of expression expected (+ 8 7 (- 8 7)) 8 ....^ ERROR
sqrt(log(2,-3))	> 13: Log argument can't be negative sqrt(log(2,-3)) ....^ ERROR
(+ log(3,5) (* 3 log(sqrt(16),2))	> 33: Expression close bracket not found (+ log(3,5) (* 3 log(sqrt(16),2)) ....^ ERROR
(+ (- (* log(sqrt(1,4)))))	> 21: Sqrt must have 1 argument (+ (- (* log(sqrt(1,4))))) ....^ ERROR

Выводы.

В ходе выполнения лабораторной работы была изучена такая структура данных как иерархические списки, а также рекурсивные методы ее обработки.

Была реализована программа на C++, использующая иерархические списки, которая анализирует строку, определяя ее соответствие префиксной записи алгебраического выражения в виде иерархического списка.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include "structs.h"
#include "LabExec.h"
#include "LabIo.h"
#include "LabExec.h"

int main (int argv, char** argc) {
    //string input = ProceedInput(argv, argc);
    string input = ReadFromConsole();
    int pos = 0;
    string copy = string(input);
    errorFlag = false;
    Expression* expression = ScanExpression(copy, pos);
    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" << endl;
    }
    else {
        cout << "SUCCESS" << endl;
    }
    return 0;
}
```

Название файла: structs.h

```
#ifndef STRUCTS_H
```

```
#define STRUCTS_H

#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <memory>
#include <random>

#include <cmath>
#include <cctype>
#include <cstring>
#include <ctime>

using namespace std;

#endif //STRUCTS_H
```

Название файла: LabIo.h

```
#ifndef SAKHAROV_LR2_LABIO_H
#define SAKHAROV_LR2_LABIO_H

#include "LabExec.h"

string ProceedInput (int argc, char *argv[]);
string ReadFromFile (std::string filename = "input.txt");
string ReadFromConsole ();
extern bool errorFlag;
extern int errorPos;
void ProceedError (const std::string & error, int pos);

#endif //LR2_LABIO_H
```

Название файла: LabIo.cpp

```
#include "LabIo.h"

bool errorFlag;
int errorPos;

// Функция ввода данных
string ProceedInput (int argc, char** argv) {
    cout << "> Lab work #1" << endl;
    if (argc > 1) {
        cout << "> Reading from argv..." << endl;
        string ins(argv[1]);
        return ins;
    }
    else {
        cout << "> Choose your input" << endl;
        cout << "> 0 - from console" << endl;
        cout << "> 1 - from file input.txt" << endl;
        cout << "> 2 - from custom file" << endl;
        int command = 0;
```

```
        cin >> command;
        string input;

        switch (command) {
            case 0:
                cout << "> Your input: ";
                return ReadFromConsole();
            case 1:
                return ReadFromFile("input.txt");
            case 2:
                cout << "> Filename: ";
                cin >> input;
                return ReadFromFile(input);
            default:
                return ProceedInput(0, nullptr);
        }
    }
}

string ReadFromFile (std::string filename) {
    ifstream infile(filename);
    if (!infile) {
        cout << "> File can't be open!" << endl;
        return "";
    }
    string res;
    infile >> res;
    return res;
}

string ReadFromConsole () {
    string res;
    getline(cin, res);
    return res;
}

void ProceedError (const std::string& error, int pos) {
    if (!errorFlag) {
        errorFlag = true;
        errorPos = pos;
        cout << "> " << pos << ": " << error << endl;
    }
    //cout << "> " << pos << ": " << error << endl;
}
```

Название файла: LabExec.h

```
#ifndef LR2_LABEXEC_H
#define LR2_LABEXEC_H

#ifndef EXPRESSIONPARSER_H
#define EXPRESSIONPARSER_H

#include "structs.h"
```

```
/*22) алгебраическое (+, -, *, sqrt(), log(,)),
 * проверка синтаксической корректности,
 * простая проверка log(,),
 * префиксная форма
 * */

enum OperatorType {
    E_PLUS,
    E_MINUS,
    E_MULTIPLY,
    E_SQRT,
    E_LOG
};

struct Expression;

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

struct OperandList {
    Operand* head;
    OperandList* tail;
};

struct Expression {
    OperatorType operatorType;
    OperandList* operandList;
};

void Skip (std::string& str, int& pos, int n);
int GetNumber (string& str, int& pos);
Operand* ScanOperand (string& input, int& pos);
OperandList* ScanOperandList (string& input, int& pos, bool isClassic);
Expression* ScanExpression (string& input, int& pos);

void CheckSqrt (OperandList* list, int& pos);
void CheckLog (OperandList* list, int& pos);
int Length (OperandList* list);

#endif //EXPRESSIONPARSER_H


#endif //LR2_LABEXEC_H
```

Название файла: LabExec.cpp

```
#include "LabExec.h"
#include "LabIo.h"
```

```
// // Удаление пробелов в начале строки
// void SkipSpaces (string& str) {
//     while (str.length() > 0 && str[0] == ' ') {
//         str = str.substr(1);
//     }
// }

// Удалить n символов из начала строки
void Skip (std::string& str, int& pos, int n) {
    if (str.length() >= n) {
        str = str.substr(n);
        //cout << str << endl;
        pos += n;
    }
}

int GetNumber (string& str, int& pos) {
    string strnum = "";
    while (isdigit(str[0]) || (strnum.length() == 0 && str[0] == '-')) {
        strnum += str[0];
        Skip(str, pos, 1);
    }
    return stoi(strnum);
}

// int|Expression
Operand* ScanOperand (string& input, int& pos) {
    //cout << "oper" << endl;
    if (input.empty() || input[0] == ')') {
        return nullptr;
    }
    Operand* operand = new Operand;
    operand->isNumber = isdigit(input[0]) || (input.length() > 1 && input[0]
== '-' && isdigit(input[1]));
    if (operand->isNumber) {
        operand->data.number = GetNumber(input, pos);
        //cout << "NUMBER " << operand->data.number << endl;
    }
    else {
        operand->data.expression = ScanExpression(input, pos);
    }
    return operand;
}

// Operand[,| ][OperandList]
OperandList* ScanOperandList (string& input, int& pos, bool isClassic) {
    //cout << "list" << endl;
    OperandList* operandList = new OperandList;
    operandList->head = ScanOperand(input, pos);
    if (operandList->head == nullptr) {
        delete operandList;
        return nullptr;
    }
```

```
        char sep = isClassic ? ',' : ' ';
        if (!input.empty() && input[0] == sep) {
            Skip(input, pos, 1);
            operandList->tail = ScanOperandList(input, pos, isClassic);
        }
        else {
            operandList->tail = nullptr;
        }
        return operandList;
    }

    // (OperatorType OperandList)
    Expression* ScanExpression (string& input, int& pos) {
        //cout << "expr" << endl;
        bool isClassic = input[0] != '(';
        Expression* expression = new Expression;
        if (isClassic) {
            if (input.find("sqrt") == 0) {
                expression->operatorType = E_SQRT;
                Skip(input, pos, 4);
            }
            else if (input.find("log") == 0) {
                expression->operatorType = E_LOG;
                Skip(input, pos, 3);
            }
            else {
                ProceedError("Function unknown operator", pos);
            }
            if (input.empty() || input[0] != '(') ProceedError("Function open
bracket not found", pos);
            Skip(input, pos, 1);
            expression->operandList = ScanOperandList(input, pos, true);
            if (expression->operandList == nullptr) ProceedError("Function
operands not found", pos);
            if (expression->operatorType == E_SQRT) {
                CheckSqrt(expression->operandList, pos);
            }
            else if (expression->operatorType == E_LOG) {
                CheckLog(expression->operandList,pos);
            }
            if (input.empty() || input[0] != ')') ProceedError("Function close
bracket not found", pos);
            Skip(input, pos, 1);
        }
        else {
            Skip(input, pos, 1);
            if (input.find('+') == 0) {
                expression->operatorType = E_PLUS;
                Skip(input, pos, 1);
            }
            else if (input.find('-') == 0) {
                expression->operatorType = E_MINUS;
                Skip(input, pos, 1);
            }
```

```
            else if (input.find('*') == 0) {
                expression->operatorType = E_MULTIPLY;
                Skip(input, pos, 1);
            }
            else {
                ProceedError("Expression unknown operator", pos);
            }
            if (input.empty() || input[0] != ' ') ProceedError("Expression
operands not found", pos);
            Skip(input, pos, 1);
            expression->operandList = ScanOperandList(input, pos, false);
            if (expression->operandList == nullptr) ProceedError("Expression
operands size = 0", pos);
            if (input.empty() || input[0] != ')') ProceedError("Expression
close bracket not found", pos);
            Skip(input, pos, 1);
        }
        return expression;
    }

    void CheckSqrt (OperandList* list, int& pos) {
        if (Length(list) == 1) {
            if (list->head->isNumber && list->head->data.number < 0) {
                ProceedError("Sqrt from negative number", pos);
            }
        }
        else {
            ProceedError("Sqrt must have 1 argument", pos);
        }
    }

    void CheckLog (OperandList* list, int& pos) {
        if (Length(list) == 2) {
            if (list->head->isNumber && (list->head->data.number == 1 ||
list->head->data.number <= 0)) {
                ProceedError("Log base can't be negative or equals 1", pos);
            }
            if (list->tail->head->isNumber && list->tail->head->data.number <=
0) {
                ProceedError("Log argument can't be negative", pos);
            }
        }
        else {
            ProceedError("Log must have 2 arguments", pos);
        }
    }

    int Length (OperandList* list) {
        OperandList* operandList = list;
        if (operandList != nullptr && operandList->head != nullptr) {
            int len = 0;
            while (operandList != nullptr) {
                len++;
                operandList = operandList->tail;
```

```
        }
        return len;
    }
    else {
        return 0;
    }
}
```