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**

Студент гр. 9381 _____ Фоминенко А.Н.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

1. Цель работы.

Ознакомиться с иерархическими списками и реализовать проверку синтаксической корректности выражения, используя иерархические списки.

2. Задание.

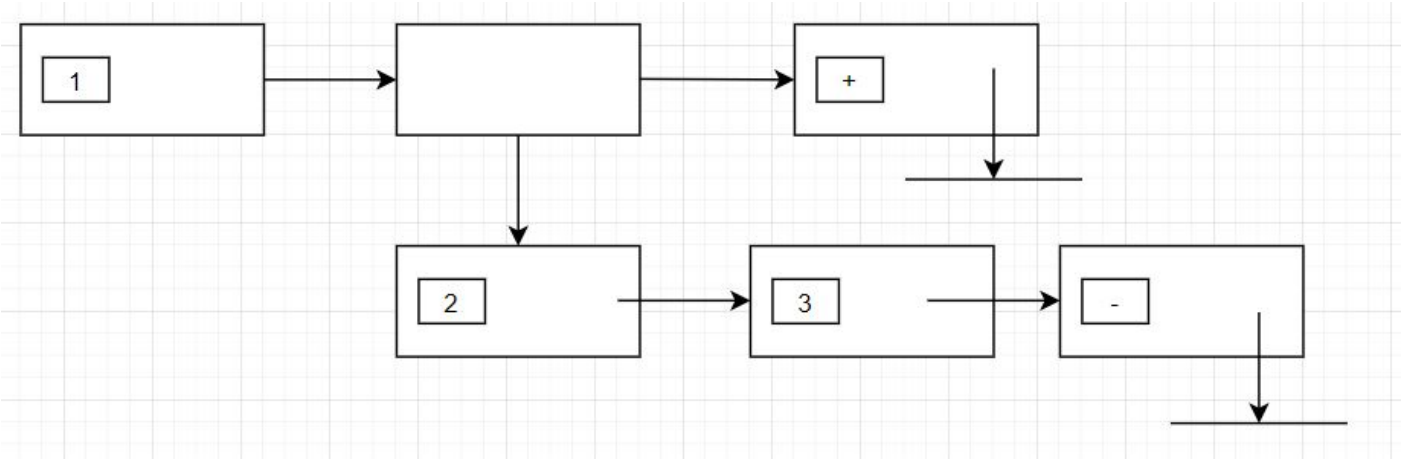
19) арифметическое, проверка синтаксической корректности и деления на 0 (простая), постфиксная форма

3. Основные теоретические положения.

Рекурсия — вызов функции (процедуры) из неё же самой непосредственно или из других функций (процедур), которые вызываются в исходной.

Иерархический список – это список, элементами которого также 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 `( 1 ( 2 3 - ) + )`



4. Описание работы алгоритма.

Программе подается на вход строка (арифметическое выражение) .

После получения строки вызывается функция `readLisp(lisp &l)` для считывания иерархического списка

После вызывается функция `void write_lisp(const lisp x, int indent)` для вывода иерархического списка

После вызывается функция `bool parser(lisp s)` проверка иерархического списка на корректность

После вызывается функция `double Eval(lisp s)` подсчет значения арифметического выражения с проверкой деления на 0.

5. Структуры для хранения Иерархического списка.

```
enum Operation {
    PLUS = 1,
    MINUS,
    DIV,
    MULT,
    ERROR
};
```

```
class Base {
//...
public:
    bool isOp; // 1 - op, 0 - func
    union {
        double number;
        Operation op;
    } base;
};

struct b_expr;

typedef b_expr *lisp;

struct Node {
    lisp head;
    lisp tail;
};

struct b_expr {
    bool isAtom;
    union {
        Base atom;
        Node pair;
    } node;
};
```

6. Пример работы программы:

Основной тест №1:

Входные данные: `( ( ( 5 6 + ) 89 - ) 67 / )`

Выходные данные (с промежуточной информацией):

```
|START:

        |START:

                |START:
```

5

6

+

|END.

89

–

|END.

67

/

|END.

Parser result : Correct

Value : -1.16418

Дополнительное тестирование :

Номер теста	Входные данные	Результат
2	( )	Parser result : Correct Value : 0
3	(9 0 /)	Parser result : Correct Value : Division by zero !
4	((-2 -) -)	Parser result : Correct Value : -2
5	(+)	Parser result : Incorrect Value : Incorrect
6	(5 (9.6 8 -) /)	Parser result : Correct Value : 3.125
7	(8 9 0 9 *)	Parser result : Correct Value :

		Error can't count value, cause of many arguments

7. Выполнение программы:

1. Для ввода информации из файла необходимо ввести “0” на вопрос программы “print '1' - console, '0' - file”.
2. Для ввода информации из консоли необходимо ввести “1” на вопрос программы “print '1' - console, '0' - file”.

8. Описание функций:

```
/**
 * функция проверка иерархического списка на атомарность
 * @param s - иерархический список
 * @return false/true
 */
bool isAtom(lisp s)
```

```
/**
 * функция считывания иерархического списка
 * @param list - иерархический список
 */
void readLisp(lisp &list)
```

```
/**
```

```
 * функция выводящая иерархический список
 * @param x - иерархический список, indent - отступ при выводе
 * @param indent
 */
void write_lisp(const lisp x, int indent)
-------------------------------------------------------------------------
/**
 * функция возвращающая голову s
 * @param s - иерархический список
 * @return иерархический список
 */
lisp head(const lisp s)
-----------------------------------------------------------------
/**
 * функция возвращающая хвост s
 * @param s - иерархический список
 * @return иерархический список
 */
lisp tail(const lisp s)
-----------------------------------------------------------------
```

```
/**
 * функция проверки синтаксической корректности
 * @param s - иерархический список
 * @return false/true
 */
bool parser(lisp s)
--------------------------------------------------------------------------
/**
 * функция подсчета значения арифметического выражения
 * @param s - иерархический список
 * @return значение арифметического выражения double
 */
double Eval(lisp s)
```

9. Вывод:

В ходе выполнения лабораторной работы была реализована программа, которая анализирует строку рекурсивным методом, создавая иерархический список и определяя его корректность.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
//19) арифметическое, проверка синтаксической корректности и деления на 0 (простая), постфиксная форма


#include <bits/stdc++.h>


using namespace std;


enum Operation {
    PLUS = 1,
    MINUS,
    DIV,
    MULT,
    ERROR
};


class Base {
private:
    static bool isNotOperation(const string &s) {
        return !(s == "+" || s == "-" || s == "/" || s == "*");
    }


    static Operation readOperation(const string &s) {
        if (s == "+") return PLUS;
        else if (s == "-") return MINUS;
        else if (s == "/") return DIV;
        else if (s == "*") return MULT;
        else return ERROR;
    }
```

```
public:
    bool isOp; // 1 - op, 0 - func
    union {
        double number;
        Operation op;
    } base;

    friend istream &operator>>(istream &input, Base &b);

    friend ostream &operator<<(ostream &out, const Base &b);

};

istream &operator>>(istream &input, Base &b) {
    string s;
    input >> s;
    if (Base::isNotOperation(s)) {
        std::istringstream is(s);
        is >> b.base.number;
        b.isOp = false;
    } else {
        b.base.op = Base::readOperation(s);
        b.isOp = true;
    }
    return input;
}

ostream &operator<<(ostream &out, const Base &b) {
    if (!b.isOp) {
        cout << b.base.number;
```

```
    } else {
        vector<string> functions = {"?", "+", "-", "/", "*"};
        cout << functions[b.base.op];
    }
    return out;
}

struct b_expr;

typedef b_expr *lisp;

struct Node {
    lisp head;
    lisp tail;
};

struct b_expr {
    bool isAtom;
    union {
        Base atom;
        Node pair;
    } node;
};
/**
 * функция проверка иерархического списка на атомарность
 * @param s - иерархический список
 * @return false/true
 */
bool isAtom(lisp s) {
    if (s == nullptr) return false;
    else return (s->isAtom);
```

```
}

lisp cons(lisp head, lisp tail) {
    lisp p;
    if (isAtom(tail)) {
        cerr << "Error: Cons(*, atom)\n";
        exit(1);
    } else {
        p = new b_expr;
        p->isAtom = false;
        p->node.pair.head = head;
        p->node.pair.tail = tail;
        return p;
    }

}




lisp makeAtom(const Base b) {
    lisp s = new b_expr;
    s->isAtom = true;
    s->node.atom = b;
    return s;
}

void readExpr(Base b, char c, lisp &list);

void readBase(Base &b, char c, bool f) {
    string s;
    if (!f) s.push_back(c);
```

```
    while (c != ' ') {
        c = (char) getchar();
        if (c != ' ')s.push_back(c);
    }
    istringstream is(s);
    is >> b;
}

void readSeq(lisp &list) {
    char c;
    Base b{};
    lisp p1, p2;

    do
        cin >> c;
    while (c == ' ');
    if (c == ')') list = nullptr;
    else {
        if (c != '(') {
            readBase(b, c, c == ')');
        }
        readExpr(b, c, p1);
        readSeq(p2);
        list = cons(p1, p2);
    }

}

void readExpr(Base b, char c, lisp &list) {
    if ((b.base.op == ERROR && b.isOp) || c == ')') {
        cerr << " ! List.Error 1 " << endl;
```

```
        exit(1);
    } else if (c != '(') list = makeAtom(b);
    else readSeq(list);
}

/**
 * функция считывания иерархического списка
 * @param list - иерархический список
 */

void readLisp(lisp &list) {
    char c;
    do
        cin >> c;
    while (c == ' ');
    Base b{};
    if (c != ')') {
        readBase(b, c, c == ')');
    }
    readExpr(b, c, list);

}

void write_seq(lisp x, int indent);

/**
 * функция выводящая иерархический список
 * @param x - иерархический список, indent - отступ при выводе
 * @param indent
 */
```

```
void write_lisp(const lisp x, int indent) {
    if (x == nullptr)return;
    for (int i = 0; i < indent; i++) cout << ' ';

    if (isAtom(x)) cout << x->node.atom << '\n';
    else {
        cout << "|START:\n";
        write_seq(x, indent + 4);
        for (int i = 0; i < indent; i++) cout << ' ';
        cout << "|END.\n";
    }
}

void write_seq(const lisp x, int indent) {
    if (x != nullptr) {
        write_lisp(x->node.pair.head, indent + 4);
        write_seq(x->node.pair.tail, indent);
    }
}



double F_val(lisp s) {
    return s->node.atom.base.number;
}

/**
 * функция возвращающая голову s
 * @param s - иерархический список
 * @return иерархический список
 */
```

```
lisp head(const lisp s) {
    if (s != nullptr)
        if (!isAtom(s)) return s->node.pair.head;
        else {
            cerr << "\nError: Head(atom) \n";
            exit(1);
        }
    else {
        cerr << "\nError: Head(nil) \n";
        exit(1);
    }
}

/**
 * функция возвращающая хвост s
 * @param s - иерархический список
 * @return иерархический список
 */

lisp tail(const lisp s) {
    if (s != nullptr)
        if (!isAtom(s)) return s->node.pair.tail;
        else {
            cerr << "\nError: Tail(atom) \n";
            exit(1);
        }
    else {
        cerr << "\nError: Tail(nil) \n";
        exit(1);
    }
}
```

```
/**
 * функция проверки синтаксической корректности
 * @param s - иерархический список
 * @return false/true
 */

bool parser(lisp s) {
    lisp a;
    if (isAtom(s))
        return true;
    else {
        if (s == nullptr || s->node.pair.head == nullptr) return true;
        //if ( + )
        if (s->node.pair.head->isAtom && s->node.pair.head->node.atom.isOp) {
            return false;
        }
        a = head(s);
        if (a->node.pair.tail == nullptr && a->isAtom && !a->node.atom.isOp) return
parser(a);
        lisp prev = s;
        while (a != nullptr && !(a->isAtom && a->node.atom.isOp)) {
            if (!parser(a)) return false;
            a = head(tail(prev));
            prev = tail(prev);
        }
    }
    return true;
}

/**
 * функция подсчета значения арифметического выражения
```

```
 * @param s - иерархический список
 * @return значение арифметического выражения double
 */

double Eval(lisp s) {
    Operation op;
    lisp a, b;
    if (s == nullptr) return 0;
    if (isAtom(s))
        return F_val(s);
    else {
        if (s->node.pair.head == nullptr) return 0;
        a = head(s);
        if (s->node.pair.tail->node.pair.head == nullptr) return Eval(a);
        b = head(tail(s));
        if (b->isAtom && b->node.atom.isOp && b->node.atom.base.op == MINUS) return
-Eval(a);
        if (b->isAtom && b->node.atom.isOp && b->node.atom.base.op != MINUS) {
            cout << "\nError can't count value, cause of 1 argument for \'" <<
b->node.atom.base.op << "\'\n";
            exit(0);
        }
        if (s->node.pair.tail->node.pair.tail->node.pair.head == nullptr ||
            !s->node.pair.tail->node.pair.tail->node.pair.head->isAtom
            || (s->node.pair.tail->node.pair.tail->node.pair.head->isAtom &&
                !s->node.pair.tail->node.pair.tail->node.pair.head->node.atom.isOp)) {
            cout << "\nError can't count value, cause of many arguments\n";
            exit(0);
        }
        op = head(tail(tail(s)))->node.atom.base.op;
        switch (op) {
            case PLUS:
```

```
                return (Eval(a) + Eval(b));
            case MINUS:
                return (Eval(a) - Eval(b));
            case DIV: {
                double b_val = Eval(b);
                if (b_val == 0) {
                    cout << "\nDivision by zero !\n";
                    exit(0);
                }
                return (Eval(a) / b_val);
            }
            case MULT:
                return (Eval(a) * Eval(b));
            default:
                return INFINITY; // ошибка
        }
    }
}

int main() {
    cout << "print '1' - console, '0' - file\n";
    char c;
    c = (char) getchar();
    if (c == '0') {
        freopen("../Test/input.txt", "r", stdin);
        freopen("../Test/output.txt", "w", stdout);
    }
    lisp l;
    readLisp(l);
    write_lisp(l, 0);
    cout << "Parser result : " << (parser(l) ? "Correct" : "Incorrect") << '\n';
```

```
    if(parser(l))
    cout << "Value : " << Eval(l) << '\n';
    else cout << "Value : Incorrect\n";
    if (c == '0') {
        fclose(stdin);
        fclose(stdout);
    } else system("pause");
    return 1;
}
// ( 89 ( 63 9 / ) - )
//  ( 8 9 0 9 * )
// ( ( -2 - ) - )
// ( 5 ( 9.6 8 - ) / )
// ( ( ( 5 6 + ) 89 - ) 67 / )
```