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: Рекурсивная обработка иерархического списка

Студент гр. 9382 _____ Субботин М. О.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы.

Познакомиться с одной из часто используемых на практике нелинейных конструкций, способами её организации и рекурсивной обработки. получить навыки решения задач обработки иерархических списков, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

Задание

20) арифметическое, упрощение, проверка деления на 0, префиксная форма

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

((x1 c1) (x2 c2) … (xk ck)),

где xi – переменная, а ci – её значение (константа).

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

Основные теоретические положения.

Требуется написать программу для упрощения арифметического выражения, записанного в префиксной форме и представленного в виде иерархического списка.

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) >

Описание и пример иерархического списка, используемого в программе

Структура иерархического списка:

```
struct s_expr;
struct  two_ptr
{
   s_expr *hd;
   s_expr *tl;
} ;  //end two_ptr;

struct nodeStr{
   string atom;
   two_ptr pair;
};

struct s_expr {
   bool tag; // true: atom, false: pair
   nodeStr node;
};                    //end s_expr

typedef s_expr *lisp;
```

struct two_ptr – структура пары.

s_expr *hd – указатель на поле, которым может быть либо атом, либо внутренний иерархический список

s_expr *tl – указатель на поле, которое будет следующим элементом списка, в этом элементе hd может также указывать либо на атом, либо на внутр. Иерархический список, т.е. все поля, которые связаны с помощью указателей tl находятся как-бы на одном уровне.

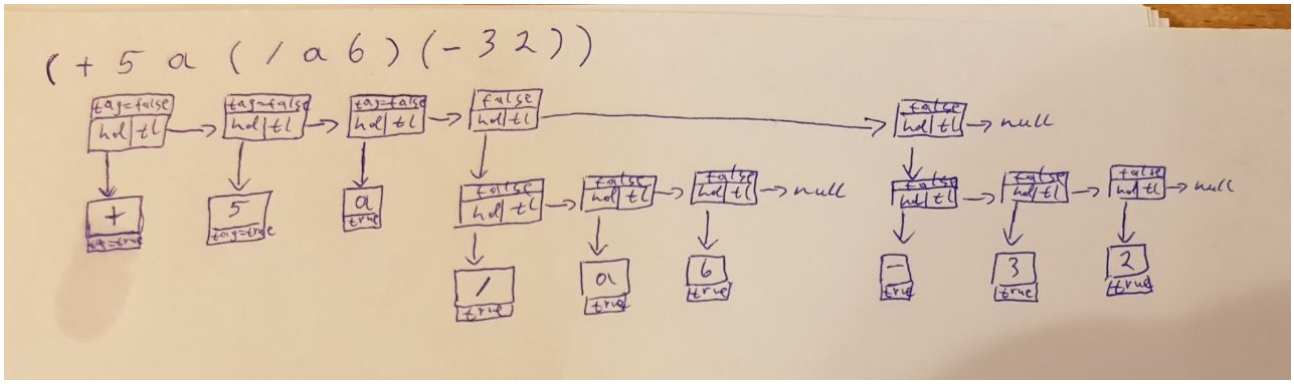
struct nodeStr – так сказать структура, которая отвечает какой тип элемента мы встречаем пару или атом.
string atom – элемент, хранящийся в атоме.

two_ptr pair – переменная, отвечающая за указатели hd и tl.

bool tag – переменная, которая отвечает за то, элемент списка атом, либо не пара.

nodeStr node – эта переменная в случае tag == true ссылается на атом, если же tag == false ссылается на пару

Пример иерархического списка, используемого в программе:



Описание основных функций:

Сигнатура: void sumEq(lisp s, bool &hasWord, bool &zeroException, int& depth)

Функция отвечает за упрощение выражения. Она вызывается рекурсивно, если в выражении встречается вложенный список.

lisp s – рассматриваемый список

Bool &hasWord – определяет, существует ли в списке константа или нет

Bool &zeroException – определяет, есть ли деление на ноль или нет

Int& depth – определяет уровень рекурсии, важна для отображения промежуточных результатов.

Описание алгоритма:

Разберем алгоритм на примере операции сложения:

Алгоритм проходится по всему списку, если элемент – это число, то он прибавляет этот элемент в общую сумму. Если элемент – это вложенный список, то мы вызываем функцию рекурсивно. Если элемент – это константа, то мы связываем указатель на эту ячейку с ячейкой, указывающую на предыдущую константу. В итоге у нас получаются связанные в начальном порядке константы и целочисленная переменная – сумма целых чисел. В зависимости от того, чему равна сумма целых чисел и сколько и какой вид имеют остальные переменные и производится упрощение.

Приведу несколько примеров упрощения, которые были произведены в программе.

1) Для операции сложения: все целые числа складываются, все константы связываются, в итоге получается выражение вида (+ 1 2 3 a 4 b 5 c) → (+ 15 a b c)
2) Для операции сложения: если в выражении все числа целые, то возвращаем полученную сумму (+ 1 2 3) → 6
3) Для операции умножения: если перемножение целых чисел равняется 0 , то общий результат будет нулевым (* 1 2 3 a b c 0) → 0
4) Для операции умножения: если перемножение целых чисел равняется 1, то общий результат будет без целых чисел (* 1 a b 1 1 c) → (* a b c)
5) Для операции деления: если числитель равняется 0 , то результат равен 0 (/ 0 5 4 3 a d c) → 0
6) Для операции деления : если знаменатель равняется 1, то в ответе будет только числитель (/ (+ a b) 1 1 1) = (+ a b)
7) Для операции вычитания : если вычитаемое равняется 0 , то в ответе будет только число из которого вычитают (- (/ a b) 0 -2 2) →(/ a b)

Тестирование.

№	Входные данные	Выходные данные
1	(- (/ a b) 0 -2 2)	(/ a b)
2	(+ 5 2 a b 0 -2 -5)	(+ a b)
3	(/ a 1 1 1)	a
4	(* a b 0)	0
5	(/ a b 0)	division by zero
6	(/ (/ a 1) 5 6 7)	(/ a 210)
7	(/ a b 5 c (/ 0 5) 10)	division by zero
8	(* a 1 a 1 b 1)	(* a a b)
9	(/ 5 e (* 5 (+ 2 -2)))	division by zero
10	(- 5 3)	2
11	(+ 5 0 3 d)	(+ 8 d)
12	(/ (+ 5 2) d)	(/ 7 d)
13	(- 5 6 (/ 5 a) 2 1 0 -4 0 (/ a 5))	(- 0 (/ 5 a) (/ a 5))

На некорректных данных

№	Входные данные	Выходные данные
14	a	a

Обработка результатов эксперимента.

Программа выдает правильные преобразования, было протестировано множество различных вариантов выражений. Для этой программы тестирования играет особую роль, так как в ней проверяется много различных вариантов. К примеру 3 тест выясняет, что надо рассматривать случай, когда знаменатель равняется 1, а 4 тест выясняет, что если умножение целых чисел дает 0, то ответ будет 0.

Выводы.

Я ознакомился с такой структурой данных, как иерархический список, научился обрабатывать ее и применять рекурсивные функции для ее обработки.

ПРИЛОЖЕНИЕ А

l_mod1.cpp

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

using namespace std;
using namespace h_list;

lisp concat(const lisp y, const lisp z);

lisp reverse(const lisp s);
lisp rev(const lisp s, const lisp z);

lisp flatten1(const lisp s);



/*
 * Эта функция производит упрощение префиксного выражения, также проверяет, есть ли деление на ноль в выражении.
 * lisp s - выражение представленное, в виде иерархического списка
 * bool & hasWord - переменная отвечает состоит ли список только из целых чисел или нет
 * int & depth - глубина рекурсии, применяется при отображении промежуточных вычислений
 *
 */
void sumEq(const lisp s, bool& hasWord, bool& zeroException, int& depth);

int main()
{
    lisp s1;
    read_lisp(s1);
    bool check = false;
    bool zeroException = false;
    int depth = 0;
    sumEq(s1,check, zeroException, depth);
    if(zeroException){
        cout << " division by zero";
    }
    else{
        cout << "\n";
```

```
            write_lisp(s1);
            cout << " - окончательный результат упрощения";
        }

        return 0;
    }


    bool isWord(string str){
        if(str[0] == '-' || str[0] == '+' || isdigit(str[0]))
            return false;
        else return true;

    }

    void shift(int depth){
        for(int i = 0; i< depth; i++){
            cout << "   ";
        }
    }

    /*
     *  Эта функция производит упрощение префиксного выражения, также проверяет, есть ли деление на ноль в выражении.
     * lisp s - выражение представленное, в виде иерархического списка
     * bool & hasWord - переменная отвечает состоит ли список только из целых чисел или нет
     *  int & depth - глубина рекурсии, применяется при отображении промежуточных вычислений
     */

    void sumEq(lisp s, bool &hasWord, bool &zeroException, int& depth){

        shift(depth);
        write_lisp(s);
        cout << " - проверка\n";

        if(zeroException)
            return;
    /* ++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
        if(s->node.pair.hd->node.atom == "+"){
            int sum = 0;
```

```
            //запоминаем начало списка
            lisp init = s;
            //создаем переменную для связывания букв и слов
            lisp temp = s;
            s = s->node.pair.tl;
            //создаем переменную для определения есть ли в списке слова или
нет
            bool wordExist = false;
            int notNumberCounter = 0;
            while(s!=nullptr){
                //мы наткнулись на атом
                if(isAtom(s->node.pair.hd)){
                    //если атом содержит букву связываем node от предыдущей
буквы с этой
                    if(isWord(s->node.pair.hd->node.atom)){
                        wordExist = true;
                        temp->node.pair.tl = s;
                        temp = s;
                        notNumberCounter++;
                    }
                    //если это число то просто считаем сумму
                    else{
                        sum+=stoi(s->node.pair.hd->node.atom);
                    }
                }
                //если же это не атом и не слово, то считаем сумму внутреннего
списка
                else{
                    bool check = false;
                    bool checkZero = false;
                    depth++;
                    sumEq(s->node.pair.hd,check,checkZero, depth);
                    depth--;
                    zeroException = checkZero;
                    if(checkZero){
                        shift(depth);
                        write_lisp(init);
                        cout << " - деление на ноль";
                        return;
                    }

                    if(!check) {
```

```
                        sum += stoi(s->node.pair.hd->node.atom);
                    } else {
                        notNumberCounter++;
                        temp->node.pair.tl = s;
                        temp = s;
                    }
                }
                s = s->node.pair.tl;
            }
            bool returnedInnerList = false;
            if(notNumberCounter >0 ){
                //если сумма не ноль то тогда у нас будет вид ( + sum b a c
... )
                if(sum != 0){
                    lisp sumhd = make_atom(to_string(sum));
                    lisp suml = cons(sumhd,init->node.pair.tl);
                    init->node.pair.tl = suml;
                    temp->node.pair.tl = nullptr;
                }
                // если сумма ноль, тогда может быть два варианта, либо буква
в последовательности одна и мы возвращаем ее
                // либо же там не одна буква и придется возвращать (+ b b b ..
)
                else {
                    if(notNumberCounter == 1){
                        // если не число - это атом
                        if(isAtom(init->node.pair.tl->node.pair.hd)){
                            init->tag=true;
                            init->node.atom       =       init->node.pair.tl-
>node.pair.hd->node.atom;
                        }
                        // если не число - это еще один список
                        else{
                            init->node.pair.hd      =      init->node.pair.tl-
>node.pair.hd->node.pair.hd;
                            init->node.pair.tl      =      init->node.pair.tl-
>node.pair.hd->node.pair.tl;
                            returnedInnerList = true;
                        }
                    }
                    else {
                        temp->node.pair.tl = nullptr;
```

```
                    }
                }

            }
            else{
                init->tag = true;
                init->node.atom = to_string(sum);
            }
            hasWord = wordExist;
            shift(depth);
            write_lisp(init);
            if(returnedInnerList){
                cout << " - конец проверки, возвращен внутренний список \n";
            }
            else {
                if (!isAtom(init)) {
                    if   (!isWord(init->node.pair.tl->node.pair.hd->node.atom))
{
                        cout << " - конец проверки, целочисленная сумма = " <<
sum
                             << " сумма констант = ";
                        write_seq(init->node.pair.tl->node.pair.tl);
                        cout << "\n";
                    } else {
                        cout << " - конец проверки, сумма констант = ";
                        write_seq(init->node.pair.tl);
                        cout << "\n";
                    }
                } else {
                    if (!isWord(init->node.atom)) {
                        cout << " - конец проверки, результат - целое число
\n";
                    } else {
                        cout << " - конец проверки, результат - константа \n";
                    }
                }
            }
        }
        /*
**************************************************************************
**** */
        else if (s->node.pair.hd->node.atom == "*"){
```

```
            int sum = 1;
            //запоминаем начало списка
            lisp init = s;
            //создаем переменную для связывания букв и слов
            lisp temp = s;
            s = s->node.pair.tl;
            //создаем переменную для определения есть ли в списке слова или
нет
            bool wordExist = false;
            //также счетчик, определяющий сколько получается в строке не чисел
            int notNumberCounter = 0;
            // переменная для определения, есть ли в последовательности ноль
            bool isZero =false;

            while(!isZero && s!=nullptr) {
                    //мы наткнулись на атом
                    if (isAtom(s->node.pair.hd)) {
                        //если атом содержит букву связываем node от
предыдущей буквы с этой
                        if (isWord(s->node.pair.hd->node.atom)) {
                            wordExist = true;
                            temp->node.pair.tl = s;
                            temp = s;
                            notNumberCounter++;
                        }
                        //если это число то проверяем не ноль ли
                        else {
                            //если это ноль
                            if(stoi(s->node.pair.hd->node.atom) == 0) {
                                sum = 0;
                                wordExist = false;
                                notNumberCounter = 0;
                                isZero = true;
                            }
                            //если не ноль то просто прибавляем это число
                            else {
                                sum *= stoi(s->node.pair.hd->node.atom);
                            }
                        }
                    }
                    //если же это не атом и не слово, то считаем сумму
внутреннего списка
```

```
                    else {
                        bool check = false;
                        bool checkZero = false;
                        depth++;
                        sumEq(s->node.pair.hd, check, checkZero, depth);
                        depth--;
                        zeroException = checkZero;
                        if(checkZero){
                            shift(depth);
                            write_lisp(init);
                            cout << " - деление на ноль \n";
                            return;
                        }
                        //если внутренний список превратился в число
                        if (!check) {
                            //если это число ноль
                            if(stoi(s->node.pair.hd->node.atom) == 0) {
                                sum = 0;
                                wordExist = false;
                                notNumberCounter = 0;
                                isZero = true;
                            }
                                //если не ноль то просто прибавляем это число
                            else {
                                sum *= stoi(s->node.pair.hd->node.atom);
                            }
                        }
                        //если внутренний список содержит несокращающиеся
символы то связываем с предыдущим таким же
                        else {
                            notNumberCounter++;
                            temp->node.pair.tl = s;
                            temp = s;
                        }
                    }
                    s = s->node.pair.tl;

                }
                bool returnedInnerList = false;
                    if (notNumberCounter > 0) {
                        //если сумма не 1 то тогда у нас будет вид ( * sum b a c
... )
```

```
                    if (sum != 1) {
                        lisp sumhd = make_atom(to_string(sum));
                        lisp suml = cons(sumhd, init->node.pair.tl);
                        init->node.pair.tl = suml;
                        temp->node.pair.tl = nullptr;
                    }
                        // если сумма один, тогда может быть два варианта,
либо буква в последовательности одна и мы возвращаем ее
                        // либо же там не одна буква и придется возвращать (+
b b b .. )
                    else {
                        if (notNumberCounter == 1) {
                            // если не число - это атом
                            if (isAtom(init->node.pair.tl->node.pair.hd)) {
                                init->tag = true;
                                init->node.atom      =      init->node.pair.tl-
>node.pair.hd->node.atom;
                            }
                                // если не число - это еще один список
                            else {
                                init->node.pair.hd    =    init->node.pair.tl-
>node.pair.hd->node.pair.hd;
                                init->node.pair.tl    =    init->node.pair.tl-
>node.pair.hd->node.pair.tl;
                                returnedInnerList = true;
                            }
                        } else {
                            temp->node.pair.tl = nullptr;
                        }
                    }


                } else {
                    init->tag = true;
                    init->node.atom = to_string(sum);
                }
                hasWord = wordExist;
            shift(depth);
            write_lisp(init);
            if(returnedInnerList){
                cout << " - конец проверки, возвращен внутренний список \n";
            }
```

```
                else {
                    if (!isAtom(init)) {
                        if   (!isWord(init->node.pair.tl->node.pair.hd->node.atom))
{
                            cout << " - конец проверки, целочисленное слагаемое =
" << sum
                                 << " слагаемое констант = ";
                            write_seq(init->node.pair.tl->node.pair.tl);
                            cout << "\n";
                        } else {
                            cout << " - конец проверки, слагаемые константы = ";
                            write_seq(init->node.pair.tl);
                            cout << "\n";
                        }
                    } else {
                        if (!isWord(init->node.atom)) {
                            cout << " - конец проверки, результат - целое число
\n";
                        } else {
                            cout << " - конец проверки, результат - константа \n";
                        }
                    }
                }

            }

            /*
////////////////////////////////////////////////////////////////////// */
            else if (s->node.pair.hd->node.atom == "/"){
                int sum = 1;
                //запоминаем начало списка
                lisp init = s;
                //создаем переменную для связывания букв и слов
                lisp temp = s->node.pair.tl;
                // числитель
                lisp numenator = s->node.pair.tl->node.pair.hd;

                s = s->node.pair.tl->node.pair.tl;
                //создаем переменную для определения есть ли в списке слова или
нет
                bool wordExist = false;
                //также счетчик, определяющий сколько получается в строке не чисел
```

```
            int notNumberCounter = 0;
            // переменная для определения, есть ли в последовательности ноль
            bool isZero =false;

            bool numenatorIsWord = false;
            bool numenatorIsZero = false;
            //simplifyNumenator
            if(!isAtom(numenator)){
                depth++;
                sumEq(numenator,numenatorIsWord, numenatorIsZero, depth);
                depth--;
            }

            if(!isAtom(numenator) || (isAtom(numenator) && isWord(numenator-
>node.atom)))
                numenatorIsWord = true;

            zeroException = numenatorIsZero;
            if(zeroException)
                return;


            // нужно все равно считать, даже если числитель ноль, потому что
может быть и деление на ноль где-то там
            while(!isZero && s!=nullptr) {
                //мы наткнулись на атом
                if (isAtom(s->node.pair.hd)) {
                    //если атом содержит букву связываем node от предыдущей
буквы с этой
                    if (isWord(s->node.pair.hd->node.atom)) {
                        wordExist = true;
                        temp->node.pair.tl = s;
                        temp = s;
                        notNumberCounter++;
                    }
                        //если это число то проверяем не ноль ли
                    else {
                        //если это ноль
                        if(stoi(s->node.pair.hd->node.atom) == 0) {
                            sum = 0;
                            wordExist = false;
                            notNumberCounter = 0;
```

```
                        isZero = true;
                    }
                        //если не ноль то просто прибавляем это число
                    else {
                        sum *= stoi(s->node.pair.hd->node.atom);
                    }
                }
            }
                //если же это не атом и не слово, то считаем сумму
внутреннего списка
            else {
                bool anyNonNumbersExist = false;
                bool checkZero = false;
                depth++;
                sumEq(s->node.pair.hd,   anyNonNumbersExist,   checkZero,
depth);
                depth--;
                zeroException = checkZero;
                if(checkZero){
                    shift(depth);
                    write_lisp(init);
                    cout << " - деление на ноль \n";
                    return;
                }
                //если внутренний список превратился в число
                if (!anyNonNumbersExist) {
                    //если это число ноль
                    if(stoi(s->node.pair.hd->node.atom) == 0) {
                        sum = 0;
                        wordExist = false;
                        notNumberCounter = 0;
                        isZero = true;
                    }
                        //если не ноль то просто прибавляем это число
                    else {
                        sum *= stoi(s->node.pair.hd->node.atom);
                    }
                }
                    //если внутренний список содержит несокращающиеся
символы то связываем с предыдущим таким же
                else {
                    notNumberCounter++;
```

```
                    temp->node.pair.tl = s;
                    temp = s;
                }
            }
            s = s->node.pair.tl;
        }

        if(sum == 0){
            zeroException = true;
            shift(depth);
            write_lisp(init);
            cout << " -деление на ноль \n";
            return;
        }

        bool returnedInnerList = false;
        if (notNumberCounter > 0) {
                // если числитель это атом
                if (isAtom(numenator)) {
                    //и если числитель - это слово
                    if (isWord(numenator->node.atom)) {
                        if(sum!= 1) {
                            lisp sumhd = make_atom(to_string(sum));
                            lisp  suml  =  cons(sumhd,  init->node.pair.tl-
>node.pair.tl);
                            init->node.pair.tl->node.pair.tl = suml;
                        }
                        temp->node.pair.tl = nullptr;
                    }
                        //если числитель - это число
                    else {
                        //если числитель был равен нулю, то ответ 0, не
смотря на то, что есть слова
                        if(stoi(numenator->node.atom) / sum == 0){
                            init->tag = true;
                            init->node.atom = "0";
                        }
                        //иначе просто упрощаем числитель
                        else {
                            numenator->node.atom                          =
to_string(stoi(numenator->node.atom) / sum);
                            temp->node.pair.tl = nullptr;
```

```
                        }
                    }
                }
                //если у нас какой-то вложенный список на месте числителя
то делаем то же самое что делали со словом
                else{
                    if(sum!=1) {
                        lisp sumhd = make_atom(to_string(sum));
                        lisp   suml   =   cons(sumhd,   init->node.pair.tl-
>node.pair.tl);
                        init->node.pair.tl->node.pair.tl = suml;
                    }
                    temp->node.pair.tl = nullptr;
                }
            }
            // если знаменатель - это число
            else {
                // если числитель - это атом
                if(isAtom(numenator)){
                    // если числитель - это слово
                    if(isWord(numenator->node.atom)){
                        if(sum!=1) {
                            lisp sumhd = make_atom(to_string(sum));
                            lisp  suml  =  cons(sumhd,  init->node.pair.tl-
>node.pair.tl);
                            init->node.pair.tl->node.pair.tl = suml;
                            suml->node.pair.tl = nullptr;
                        }
                        else {
                            init->tag = true;
                            init->node.atom      =      init->node.pair.tl-
>node.pair.hd->node.atom;

                        }
                    }
                    // если числитель - это число
                    else {
                        init->tag = true;
                        init->node.atom     =     to_string(stoi(numenator-
>node.atom) / sum);
                    }
                }
```

```
                    // если числитель - это внутренний список
                    else {
                        if(sum != 1) {
                            lisp sumhd = make_atom(to_string(sum));
                            lisp   suml   =   cons(sumhd,   init->node.pair.tl-
>node.pair.tl);
                            init->node.pair.tl->node.pair.tl = suml;
                            suml->node.pair.tl = nullptr;
                        }
                        else {
                            init->node.pair.hd      =      init->node.pair.tl-
>node.pair.hd->node.pair.hd;
                            init->node.pair.tl      =      init->node.pair.tl-
>node.pair.hd->node.pair.tl;
                            returnedInnerList = true;
                        }
                    }
                }
                hasWord = wordExist || numenatorIsWord;
                shift(depth);
                write_lisp(init);
                if(returnedInnerList){
                    cout << " - конец проверки, возвращен внутренний список \n";
                }
                else {
                    if (!isAtom(init)) {
                        if   (!isWord(init->node.pair.tl->node.pair.hd->node.atom))
{
                            cout << " - конец проверки, целочисленный числитель =
"
                                 << init->node.pair.tl->node.pair.hd->node.atom
                                 << " знаменатель = ";
                            write_seq(init->node.pair.tl->node.pair.tl);
                            cout << "\n";
                        } else {
                            if (isAtom(init->node.pair.tl->node.pair.hd)) {
                                cout << " - конец проверки, константный числитель
= "
                                     <<          init->node.pair.tl->node.pair.hd-
>node.atom
                                     << " знаменатель = ";
                            } else {
```

```
                                cout << " - конец проверки, константный числитель
= ";
                                write_lisp(init->node.pair.tl->node.pair.hd);
                                cout << " знаменатель = ";
                            }
                            write_seq(init->node.pair.tl->node.pair.tl);
                            cout << "\n";
                        }
                    } else {
                        if (!isWord(init->node.atom)) {
                            cout << " - конец проверки, результат - целое число
\n";
                        } else {
                            cout << " - конец проверки, результат - константа \n";
                        }
                    }
                }

            }
        /* ------------------------------------------------------------------ */
            else if (s->node.pair.hd->node.atom == "-") {

                int sum = 0;
                //запоминаем начало списка
                lisp init = s;
                //создаем переменную для связывания букв и слов
                lisp temp = s->node.pair.tl;
                // числитель
                lisp numenator = s->node.pair.tl->node.pair.hd;

                s = s->node.pair.tl->node.pair.tl;

                //создаем переменную для определения есть ли в списке слова или
нет
                bool wordExist = false;
                //также счетчик, определяющий сколько получается в строке не чисел
                int notNumberCounter = 0;
                // переменная для определения, есть ли в последовательности ноль
                bool isZero = false;

                bool numenatorIsWord = false;
                bool numenatorIsZero = false;
```

```
            //simplifyNumenator
            if (!isAtom(numenator)) {
                depth++;
                sumEq(numenator, numenatorIsWord, numenatorIsZero, depth);
                depth--;
            }

            if (!isAtom(numenator) || (isAtom(numenator) && isWord(numenator-
>node.atom)))
                numenatorIsWord = true;

            zeroException = numenatorIsZero;
            if (zeroException)
                return;


            while (!isZero && s != nullptr) {
                //мы наткнулись на атом
                if (isAtom(s->node.pair.hd)) {
                    //если атом содержит букву связываем node от предыдущей
буквы с этой
                    if (isWord(s->node.pair.hd->node.atom)) {
                        wordExist = true;
                        temp->node.pair.tl = s;
                        temp = s;
                        notNumberCounter++;
                    }
                        //если это число то суммируем
                    else {
                        sum += stoi(s->node.pair.hd->node.atom);
                    }
                }
                    //если же это не атом и не слово, то считаем сумму
внутреннего списка
                else {
                    bool anyNonNumbersExist = false;
                    bool checkZero = false;
                    depth++;
                    sumEq(s->node.pair.hd,   anyNonNumbersExist,   checkZero,
depth);
                    depth--;
```

```
                    zeroException = checkZero;
                    if (checkZero) {
                        shift(depth);
                        write_lisp(init);
                        cout << " - деление на ноль \n";
                        return;
                    }
                    //если внутренний список превратился в число
                    if (!anyNonNumbersExist) {
                        //просто прибавляем это число
                        sum += stoi(s->node.pair.hd->node.atom);
                    }
                        //если внутренний список содержит несокращающиеся
символы то связываем с предыдущим таким же
                    else {
                        notNumberCounter++;
                        temp->node.pair.tl = s;
                        temp = s;
                    }
                }
                s = s->node.pair.tl;
            }

            bool returnedInnerList = false;
            if (notNumberCounter > 0) {
                // если числитель это атом
                if (isAtom(numenator)) {
                    //и если числитель - это слово
                    if (isWord(numenator->node.atom)) {
                        if (sum != 0) {
                            lisp sumhd = make_atom(to_string(sum));
                            lisp   suml   =   cons(sumhd,   init->node.pair.tl-
>node.pair.tl);
                            init->node.pair.tl->node.pair.tl = suml;
                        }
                        temp->node.pair.tl = nullptr;
                    }
                        //если числитель - это число
                    else {
                        numenator->node.atom    =    to_string(stoi(numenator-
>node.atom) - sum);
                        temp->node.pair.tl = nullptr;
```

```
                    }
                }
                    //если у нас какой-то вложенный список на месте числителя
то делаем то же самое что делали со словом
                else {
                    if (sum != 0) {
                        lisp sumhd = make_atom(to_string(sum));
                        lisp    suml    =    cons(sumhd,    init->node.pair.tl-
>node.pair.tl);
                        init->node.pair.tl->node.pair.tl = suml;
                    }
                    temp->node.pair.tl = nullptr;
                }
            }
                // если знаменатель - это число
            else {
                // если числитель - это атом
                if (isAtom(numenator)) {
                    // если числитель - это слово
                    if (isWord(numenator->node.atom)) {
                        if (sum != 0) {
                            lisp sumhd = make_atom(to_string(sum));
                            lisp   suml   =   cons(sumhd,   init->node.pair.tl-
>node.pair.tl);
                            init->node.pair.tl->node.pair.tl = suml;
                            suml->node.pair.tl = nullptr;
                        } else {
                            init->tag = true;
                            init->node.atom       =       init->node.pair.tl-
>node.pair.hd->node.atom;
                        }
                    }
                        // если числитель - это число
                    else {
                        init->tag = true;
                        init->node.atom = to_string(stoi(numenator->node.atom)
- sum);
                    }
                }
                    // если числитель - это внутренний список
                else {
                    if (sum != 0) {
```

```
                        lisp sumhd = make_atom(to_string(sum));
                        lisp    suml    =    cons(sumhd,    init->node.pair.tl-
>node.pair.tl);
                        init->node.pair.tl->node.pair.tl = suml;
                        suml->node.pair.tl = nullptr;
                    } else {
                        init->node.pair.hd = init->node.pair.tl->node.pair.hd-
>node.pair.hd;
                        init->node.pair.tl = init->node.pair.tl->node.pair.hd-
>node.pair.tl;
                        returnedInnerList = true;
                    }

                }
            }

            hasWord = wordExist || numenatorIsWord;

            shift(depth);
            write_lisp(init);
            if (returnedInnerList) {
                cout << " - конец проверки, возвращен внутренний список \n";
            } else {
                if (!isAtom(init)) {
                    if   (!isWord(init->node.pair.tl->node.pair.hd->node.atom))
{
                        cout << " - конец проверки, целочисленное уменьшаемое
= "
                             << init->node.pair.tl->node.pair.hd->node.atom
                             << " вычитаемое = ";
                        write_seq(init->node.pair.tl->node.pair.tl);
                        cout << "\n";
                    } else {
                        if (isAtom(init->node.pair.tl->node.pair.hd)) {
                            cout   <<   "   -   конец   проверки,   константное
уменьшаемое = "
                                 <<          init->node.pair.tl->node.pair.hd-
>node.atom
                                 << " вычитаемое = ";
                        } else {
                            cout   <<   "   -   конец   проверки,   константный
уменьшаемое = ";
```

```
                            write_lisp(init->node.pair.tl->node.pair.hd);
                            cout << " вычитаемое = ";
                        }
                        write_seq(init->node.pair.tl->node.pair.tl);
                        cout << "\n";
                    }
                } else {
                    if (!isWord(init->node.atom)) {
                        cout << " - конец проверки, результат - целое число
\n";
                    } else {
                        cout << " - конец проверки, результат - константа \n";
                    }
                }
            }
        }

    }




    //.........................
    lisp concat(const lisp y, const lisp z)
    {
        if (isNull(y))
            return copy_lisp(z);
        else
            return cons(copy_lisp(head(y)), concat(tail(y), z));
    } // end concat
    // ----------------------
    lisp reverse(const lisp s)
    {
        return (rev(s, NULL));
    }
    //.......................
    lisp rev(const lisp s, const lisp z)
    {
        if (isNull(s))
            return (z);
        else if (isAtom(head(s)))
```

```
        return (rev(tail(s), cons(head(s), z)));
    else
        return (rev(tail(s), cons(rev(head(s), NULL), z)));
}
//.....................
lisp flatten1(const lisp s)
{
    if (isNull(s))
        return NULL;
    else if (isAtom(s))
        return cons(make_atom(getAtom(s)), NULL);
    else // s - Г-ГЃГЇГіГ±ГІГ®Г© Г±ГЇГЁГ±Г®ГЄ
    if (isAtom(head(s)))
        return cons(make_atom(getAtom(head(s))), flatten1(tail(s)));
    else // Not Atom(Head(s))
        return concat(flatten1(head(s)), flatten1(tail(s)));
} // end flatten1
```

L_intrfc.h

```
#ifndef ALGOLIST_L_INTRFC_H
#define ALGOLIST_L_INTRFC_H

#include <string>
#include <variant>
using namespace std;

// интерфейс АТД "Иерархический Список"
namespace h_list
{
    struct s_expr;
    struct  two_ptr
    {
        s_expr *hd;
        s_expr *tl;
    } ; //end two_ptr;

    struct nodeStr{
        string atom;
        two_ptr pair;
    };

    struct s_expr {
        bool tag; // true: atom, false: pair
```

```
        nodeStr node;
    };          //end s_expr

    typedef s_expr *lisp;

// функции
    void print_s_expr( lisp s );
    // базовые функции:
    lisp head (const lisp s);
    lisp tail (const lisp s);
    lisp cons (const lisp h, const lisp t);
    lisp make_atom (const string x);
    bool isAtom (const lisp s);
    bool isNull (const lisp s);
    void destroy (lisp s);

    string getAtom (const lisp s);

    // функции ввода:
    void read_lisp ( lisp& y);          // основная
    void read_s_expr (string prev, lisp& y, string str, int &idx);
    void read_seq ( lisp& y, string str, int &idx);

    // функции вывода:
    void write_lisp (const lisp x);     // основная
    void write_seq (const lisp x);

    lisp copy_lisp (const lisp x);

} // end of namespace h_list

#endif //ALGOLIST_L_INTRFC_H
```

L_impl.cpp

```
// continue of namespace h_list
#include "l_intrfc.h"
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;
namespace h_list
```

```
{

//.................................
    lisp head (const lisp s)
    {// PreCondition: not null (s)
        if (s != NULL) if (!isAtom(s))  return s->node.pair.hd;
            else { cerr << "Error: Head(atom) \n"; exit(1); }
        else { cerr << "Error: Head(nil) \n";
            exit(1);
        }
    }
//.................................
    bool isAtom (const lisp s)
    {   if(s == NULL) return false;
        else return (s -> tag);
    }
//.................................
    bool isNull (const lisp s)
    { return s==NULL;
    }
//.................................
    lisp tail (const lisp s)
    {// PreCondition: not null (s)
        if (s != NULL) if (!isAtom(s))  return s->node.pair.tl;
            else { cerr << "Error: Tail(atom) \n"; exit(1); }
        else { cerr << "Error: Tail(nil) \n";
            exit(1);
        }
    }
//.................................
    lisp cons (const lisp h, const lisp t)
    // PreCondition: not isAtom (t)
    {lisp p;
        if (isAtom(t)) { cerr << "Error: Tail(nil) \n"; exit(1);}
        else {
            p = new s_expr;
            if ( p == NULL) {cerr << "Memory not enough\n"; exit(1); }
            else {
                p->tag = false;
                p->node.pair.hd = h;
                p->node.pair.tl = t;
                return p;
```

```
                }
            }
        }
    //.........................
        lisp make_atom (const string x)
        {   lisp s;
            s = new s_expr;
            s -> tag = true;
            s->node.atom = x;
            return s;
        }

    //.........................
        void destroy (lisp s)
        {
            if ( s != NULL) {
                if (!isAtom(s)) {
                    destroy ( head (s));
                    destroy ( tail(s));
                }
                delete s;
                // s = NULL;
            };
        }
    //.........................
        string getAtom (const lisp s)
        {
            if (!isAtom(s)) { cerr << "Error: getAtom(s) for !isAtom(s) \n";
exit(1);}
            else return (s->node.atom);
        }


    //.........................
    // ввод списка с консоли
        void read_lisp ( lisp& y)
        {   string x;
            //удаляем пробелы перед первым элементом
            ifstream infile ("../in.txt");
            getline(infile,x);
            cout << x << " - выражение для упрощения \n\n";
```

```
            int i = 0;

            while(i < size(x) && x[i] == ' '){
                i++;
            }
            if(i < size(x)) {
                string res(1, x[i]);
                i++;
                read_s_expr(res, y, x, i);
            }
            else {
                std::cout << "nope";
            }
        } //end read_lisp

    //.........................
        void read_s_expr (string prev, lisp& y, string str, int& idx)
        { //prev - ранее прочитанный символ}
            //если первй символ ) то все печально
            if ( prev == ")" ) {cerr << " ! List.Error 1 " << endl; exit(1); }
            //тогда если не ( то всего один элемент может быть и запишем его в
y
            else if ( prev != "(" )
            {
                y = make_atom(prev);
            }
            //если все же последовательность...
            else read_seq (y, str, idx);
        } //end read_s_expr

    //.........................
        void read_seq ( lisp& y, string str, int& idx)
        {   char x;
            lisp p1, p2;
            if (idx == size(str)){
                cerr << " ! List.Error 2 " << endl; exit(1);
            }
                while(idx < size(str) && str[idx]==' ') {
                    idx++;
                }

                string res;
```

```
                while(idx < size(str) && str[idx]!=' '){
                    res+=str[idx];
                    idx++;
                }

                if ( res == ")" ) y = nullptr;
                else {
                    if(str[idx-1] == ')') {
                        idx -= 2;
                        res.pop_back();
                    }
                    idx++;
                    read_s_expr ( res,p1, str,idx);
                    read_seq ( p2,str,idx);
                    y = cons (p1, p2);
                }
            } //end read_seq
        //.........................
        // Процедура вывода списка с обрамляющими его скобками - write_lisp,
        //  а без обрамляющих скобок - write_seq
            void write_lisp (const lisp x)
            {//пустой список выводится как ()
                if (isNull(x)) cout << " ()";
                else if (isAtom(x)) cout << ' ' << x->node.atom;
                else  { //непустой список}
                    cout << " (" ;
                    write_seq(x);
                    cout << " )";
                }
            } // end write_lisp
        //.........................
            void write_seq (const lisp x)
            {//выводит  последовательность  элементов  списка  без  обрамляющих  его
скобок
                if (!isNull(x)) {
                    write_lisp(head (x));
                    write_seq(tail (x));
                }
            }
        //.........................
            lisp copy_lisp (const lisp x)
            {   if  (isNull(x)) return NULL;
```

```
        else if (isAtom(x)) return make_atom (x->node.atom);
        else return cons (copy_lisp (head (x)), copy_lisp (tail(x)));
    } //end copy-lisp

} // end of namespace h_list
```