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

ОТЧЁТ

по лабораторной работе №2
по дисциплине «Алгоритмы и струкруры данных»
Тема: Иерархические списки

Студент гр. 9303 _____ Эйсвальд М.И.

Преподаватель _____ Филатов Ар.Ю.

Санкт-Петербург
2020

Цель работы.

Освоить работу с иерархическими списками с помощью рекурсивных алгоритмов. Научиться решать прикладную задачу с помощью иерархических списков.

Задание.

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.

Вариант 24: алгебраическое (+, -, *, power(,)), вычисление, префиксная форма

В заданиях 22 – 24 функции sqrt(), log(,), sin(), cos(), power(,) используются в классической форме (аргумент(ы) в скобках, а слева от скобок – название функции), а не в префиксной или постфиксной!

Основные теоретические положения.

Иерархический список — структура данных, представляющая собой список, в котором каждый элемент является или атомом, или иерархическим списком. Иерархический неатомарный список удобно представлять в виде головы и хвоста, как это делалось и с линейными списками. В нашем случае голова также может быть списком, чего не может быть для линейного списка.

Выполнение работы.

Было решено использовать предоставленный интерфейс для работы со списком, изменив ввод-вывод с тем, чтобы осуществлять их как с консоли (на экран), так и из файла (в файл). С этой целью функции ввода принимают

строку по ссылке, от которой «откусывают» нужные им куски, а функции вывода — указатель на поток вывода. Поскольку в команде `power` более одного символа, базовый тип данных списка — C++-style строка.

Вводимая строчка сначала обрабатывается: удаляется пара скобок вокруг аргумента функции `power`, удаляется запятая между аргументами, добавляются пробелы вокруг скобок, чтобы не считать сразу и скобку, и аргумент. Потом из строки читается содержимое собственно списка, после чего рекурсивно вычисляется значение выражения. Значением атомарного списка считается числовое значение, записанное в его строке (при корректном вводе атомарные списки не содержат символов операций), значением списка с нулевым хвостом — значение головы, значением списка с головой-атомом, где записан символ операции, — результат применения операции к голове хвоста и хвосту хвоста.

Тестирование.

Скриншоты с результатами теста см. в приложении А.

Вывод.

Были изучены иерархические списки, решена прикладная задача с использованием списков. Результатом работы стала программа на языке C++, вычисляющая выражение с заданными операциями в префиксной форме, представляя это выражение в виде иерархического списка. Несмотря на кажущееся неудобство, разработать алгоритм программы оказалось намного легче, чем, например, обработать ввод, чтобы придать ему правильную структуру. Из этого следует, что иерархические списки действительно могут быть с успехом применены для решения прикладных задач.

ПРИЛОЖЕНИЕ А
РЕЗУЛЬТАТЫ ТЕСТОВ

Рисунок 1 – Простой тест

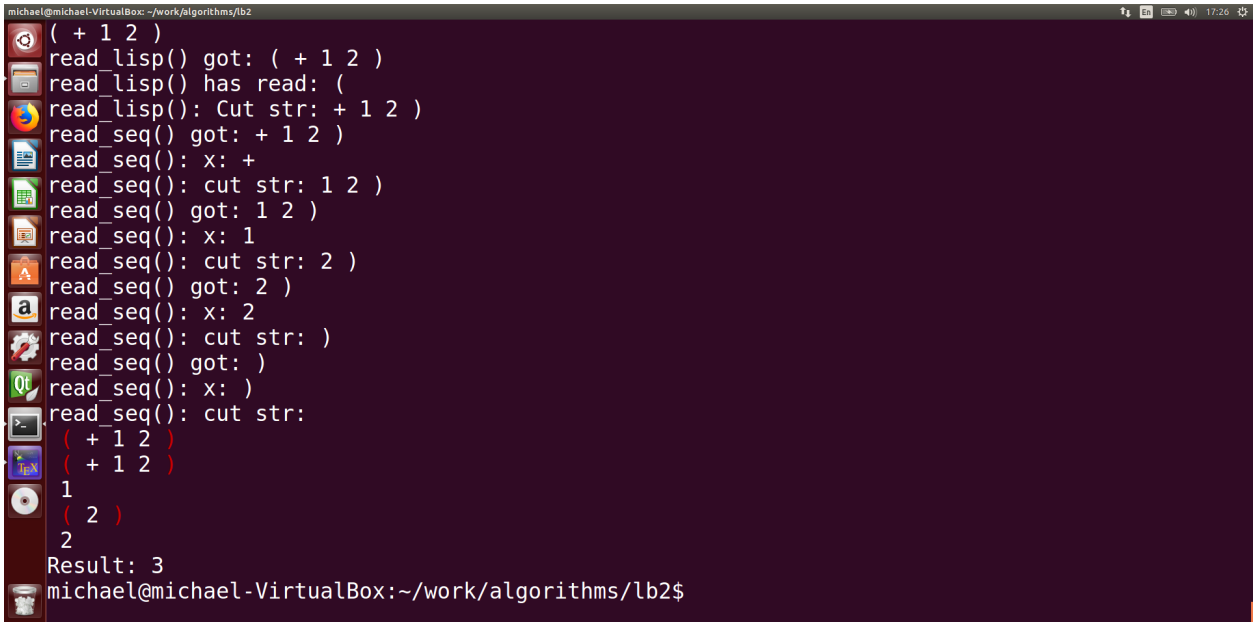

Рисунок 2 – Сложение

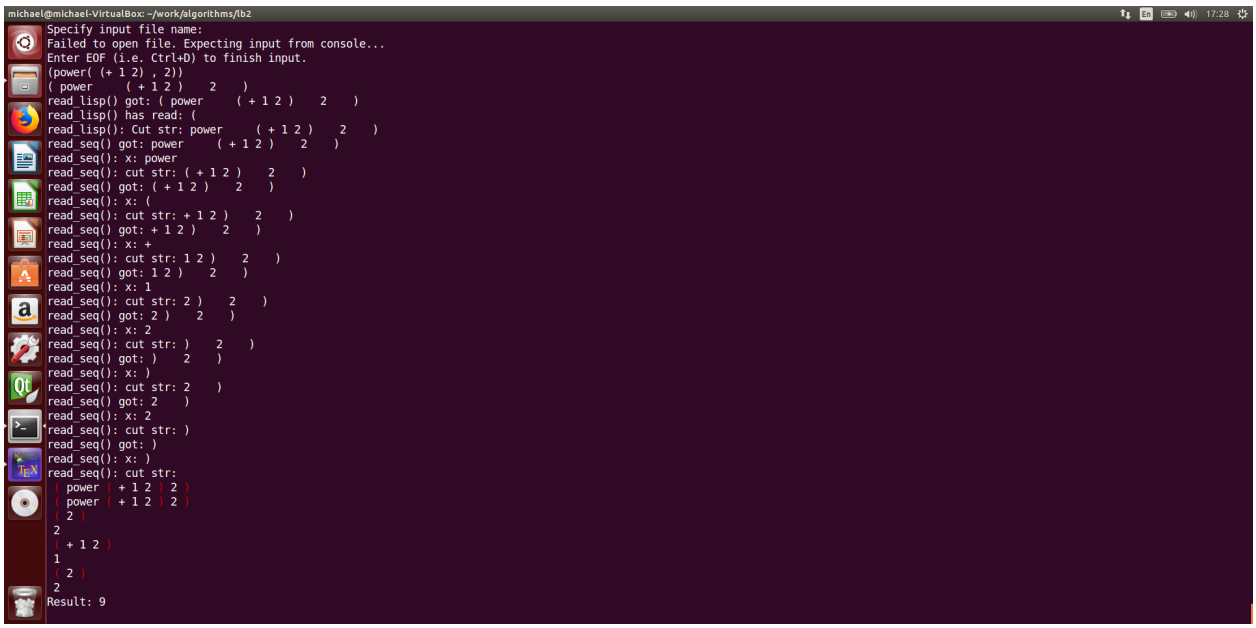

Рисунок 3 – Сложное выражение, включающее возведение в степень

Рисунок 4 – Выражение с двумя действиями

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД

l_intrfc.h:

l_impl.cpp:

```
#include ”l_intrfc.h”
#include <iostream>
#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 base 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;
};
}
//.........................
base getAtom (const lisp s)
{
if (!isAtom(s)) { cerr << ”Error: getAtom(s) for !isAtom(s) \n”; exit(1);}
else return (s->node.atom);
}


//.........................
// □□□□ □□□□□□ □ □□□□□□
void read_lisp ( lisp& y , string& str)
{ base x;
//infile->is_open() ? *infile >> x : cin >> x;
cout << ”read_lisp() got: ” << str << ”\n”;
x = str.substr(0, str.find_first_of(” ”));
cout << ”read_lisp() has read: ” << x << ”\n”;
str.erase(0, str.find_first_not_of(” ”,str.find_first_of(” ”)));
cout << ”read_lisp(): Cut str: ” << str << ”\n”;
read_s_expr (x, y, str);
} //end read_lisp
//.........................
void read_s_expr (base prev, lisp& y, string& str)
{ //prev - □□□□□ □□□□□□□□□□ □□□□□□}
if ( prev == ”)” ) {cerr << ” ! List.Error 1 ” << endl; exit(1); }
else if ( prev != ”(” ) y = make_atom (prev);
 else read_seq (y, str);
} //end read_s_expr
//.........................
void read_seq (lisp& y , string& str)
{ base x;
lisp p1, p2;

//infile->is_open() ? *infile >> x : cin >> x;
cout << ”read_seq() got: ” << str << ”\n”;
x = str.substr(0, str.find_first_of(” ”));
cout << ”read_seq(): x: ” << x << ”\n”;
str.erase(0, str.find_first_not_of(” ”,str.find_first_of(” ”)));
cout << ”read_seq(): cut str: ” << str <<  ”\n”;
if ( x == ”)” ) y = NULL;
else {
read_s_expr ( x, p1, str);
read_seq (p2, str) ;
y = cons (p1, p2);
}
} //end read_seq
//.........................
// □□□□□□□□□ □□□□□□ □□□□□□ □ □□□□□□□□□□□ □□□ □□□□□□□□ - write_lisp,
//  □ □□□ □□□□□□□□□□ □□□□□□ - write_seq
void write_lisp (const lisp x, ofstream* outfile)
{//□□□□□□ □□□□□□ □□□□□□□□ □□□ ()
if (isNull(x)){
cout << RED << ” ()” << NORMAL;
```

```
*outfile << ” ()”;
}
else if (isAtom(x)) {cout << ’ ’ << x->node.atom; *outfile << ” ” << x->node.atom;}
else  { //□□□□□□□□□ □□□□□□□}
cout << RED << ” (” << NORMAL;
*outfile << ” (”;
write_seq(x, outfile);
cout << RED << ” )” << NORMAL;
*outfile << ” )”;
}
} // end write_lisp
//.........................
void write_seq (const lisp x, ofstream* outfile)
{//□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□ □□□□□□□ □□□□ □□□□□□□□□□□□□ □□□□ □□□□□□
if (!isNull(x)) {
write_lisp(head (x), outfile);
write_seq(tail (x), outfile);
}
}
//.........................
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


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 flatten1(const lisp s)
        {
                if (isNull(s)) return NULL;
                else if(isAtom(s)) return cons(make_atom(getAtom(s)),NULL);
                else //s - □□□□□□□□□ □□□□□□□
           if (isAtom(head(s))) return cons( make_atom(getAtom(head(s))),flatten1(tail(
                else //Not Atom(Head(s))
                        return concat(flatten1(head(s)),flatten1(tail(s)));
        }

double calculate(const lisp list, ofstream* outfile){
write_lisp(list, outfile);
cout << ”\n”;
*outfile << ”\n”;
//cout << ”Tail -- null? ” << (!isNull(list) && isNull(tail(list))) << ”\n”;
if(isNull(list)) return NAN;
if(isAtom(list)) return stod(list->node.atom);
if(isNull(tail(list))) return calculate(head(list),outfile);
if(isAtom(head(list))){
if(head(list)->node.atom == ”+”){
return calculate(head(tail(list)),outfile) + calculate(tail(tail(list)),outfile);
}
if(head(list)->node.atom == ”-”){
                return calculate(head(tail(list)),outfile) - calculate(tail(tail(list)
                }
if(head(list)->node.atom == ”*”){
                return calculate(head(tail(list)),outfile) * calculate(tail(tail(list)
```

```
                }
if(head(list)->node.atom == ”power”){
                return pow(calculate(head(tail(list)),outfile),calculate(tail(tail(lis
                }

}
return NAN;
}
}
```

l_intrfc.h:

```
#include <string>
#include <cmath>
#include <fstream>

using namespace std;

#define NORMAL ”\033[0m”
#define RED ”\033[0;31m”
// □□□□□□□□□□ □□□□ ”□□□□□□□□□□□□□□□ □□□□□□□”
namespace h_list
{
typedef std::string base; // □□□□□□□□ □□□□ □□□□□□□□□□ (□□□□□□□)

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

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

typedef s_expr *lisp;

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

base getAtom (const lisp s);

double calculate(const lisp s, ofstream* outfile);
```

```
// □□□□□□□□ □□□□□□:
void read_lisp ( lisp& y, /*ifstream* infile*/ string& str); // □□□□□□□□□
void read_s_expr (base prev, lisp& y , /*ifstream* infile*/ string& str);
void read_seq ( lisp& y, /*ifstream* infile*/ string& str);

// □□□□□□□□ □□□□□□□:
void write_lisp (const lisp x, ofstream* outfile); // □□□□□□□□□
void write_seq (const lisp x, ofstream* outfile);

lisp copy_lisp (const lisp x);
lisp flatten1(const lisp s);
lisp concat (const lisp y, const lisp z);

} // end of namespace h_list
```

main.cpp:

```
#include <iostream>
#include <fstream>
#include ”l_intrfc.h”

using namespace h_list;
using namespace std;

void clear_input(string& expression, char chr){
int pos = 0;
        while(pos != string::npos){
                pos = expression.find(chr, pos);
                if(pos != string::npos){
                     expression.insert(pos+1 , ” ”);
                     expression.insert(pos,” ”);
                     pos += 2;
                }
        }
}

int main(){
lisp lst = NULL;
lisp flat_lst = NULL;
std::string expression;
std::string filename;
std::cout << ”Specify input file name: ”;
getline(cin, filename);
ifstream infile(filename.c_str());
ofstream outfile(”output.txt”);

if(!infile){
std::cout << ”Failed to open file. Expecting input from console...\n”;
std::cout << ”Enter EOF (i.e. Ctrl+D) to finish input.\n”;
}

while((!infile.eof() && infile.is_open()) || (!cin.eof() && !infile.is_open())){
infile ? getline(infile,expression) : getline(cin,expression);
if(expression != ””){
outfile << ”Initial string: \”” << expression << ”\”\n”;
while(expression.find(”,”) != string::npos){
expression[expression.find(”,”)] = ’ ’;
}
```

```
clear_input(expression, '(');
clear_input(expression, ')');
int pos = 0;
while(expression.find("power",pos) != string::npos){
pos = expression.find("power",pos) + 1;
expression[expression.find("(",pos)] = ' ';
int counter = 0;
int pos_bracket = pos;
while(1){
if(expression.find(")",pos_bracket)<
expression.find("(",pos_bracket)){
if(!counter){ expression[expression.find(")",pos_bracket)]= ' ';
   break;
}
else --counter;
}
else ++counter;
pos_bracket = min(expression.find(")",pos_bracket),
expression.find("(",pos_bracket)) + 1;
}
}
expression.erase(0, expression.find_first_not_of(" "));
cout << expression << "\n";
outfile << "Parsed string: \"" << expression << "\"\n";
read_lisp(lst, expression);
write_lisp(lst, &outfile);
         std::cout << '\n';
outfile << "\n";
int result = calculate(lst, &outfile);
cout << "Result: " << result << "\n";
outfile << "Result: " << result << "\n";
         destroy(lst);
lst = NULL;
}
}

return 0;

/*lisp lst = NULL;
read_lisp(lst, &infile);
write_lisp(lst);
std::cout << '\n';
destroy(lst);
return 0;*/
}
```