

**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

Судаков Е.В.

Преподаватель

Фирсов М.А.

Санкт-Петербург
2020

1. Цель работы.

Ознакомиться с рекурсивными алгоритмами и реализовать рекурсивную функцию обработки иерархического списка.

2. Задание.

Вариант 16.

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))).

Необходимо реализовать обработку логических выражений в префиксной форме, выполнить проверку синтаксической корректности, добавить 4-ую операцию (которая может принимать 2 аргумента).

3. Основные теоретические положения.

Согласно определению иерархического списка, структура непустого иерархического списка - это элемент размеченного объединения множества атомов и множества пар «голова-хвост».

На рисунке 1 представлен иерархический список из главного теста, демонстрирующего работу программы.

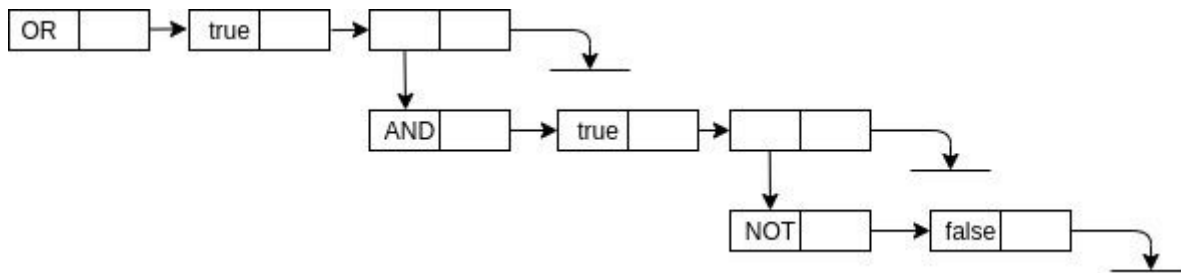


Рисунок 1. (OR true (AND true (XOR true false)))

4. Пример работы программы:

Основной тест №1:

Входные данные: (OR true (AND true (XOR true false)))

Выходные данные (с промежуточной информацией):

Read : (OR true (AND true (XOR true false)))

Parser for (OR true (AND true (XOR true false))) is called

| *Parser for OR is called*

| *OR is correct function*

| *Parser for (true (AND true (XOR true false))) is called*

| | *Parser for true is called*

| | *true is correct operand*

| | *Parser for ((AND true (XOR true false))) is called*

| | | *Parser for (AND true (XOR true false)) is called*

| | | | *Parser for AND is called*

| | | | *AND is correct function*

| | | | *Parser for (true (XOR true false)) is called*

| | | | | *Parser for true is called*

					<i>true is correct operand</i>				
					<i>Parser for ((XOR true false)) is called</i>				
						<i>Parser for (XOR true false) is called</i>			
							<i>Parser for XOR is called</i>		
							<i>XOR is correct function</i>		
							<i>Parser for (true false) is called</i>		
								<i>Parser for true is called</i>	
								<i>true is correct operand</i>	
								<i>Parser for (false) is called</i>	
									<i>Parser for false is called</i>
									<i>false is correct operand</i>
									<i>(false) is correct lisp</i>
								<i>(true false) is correct lisp</i>	
						<i>(XOR true false) is correct lisp</i>			
					<i>((XOR true false)) is correct lisp</i>				
				<i>(true (XOR true false)) is correct lisp</i>					
			<i>(AND true (XOR true false)) is correct lisp</i>						
		<i>((AND true (XOR true false))) is correct lisp</i>							
	<i>(true (AND true (XOR true false))) is correct lisp</i>								
<i>(OR true (AND true (XOR true false))) is correct lisp</i>									
<i>Lisp correct</i>									

Дополнительное тестирование :

Номер теста	Входные данные	Результат
2	()	Lisp correct
3	(AND true false)	Lisp correct
4	(NOT)	Lisp not correct
5	(true false true)	Lisp correct
6	(true AND)	Lisp not correct

4. Выполнение программы:

1. Для ввода информации из файла необходимо ввести “1” на вопрос программы “Строка из консоли или из файла (0/1)?”.
2. Для ввода информации через консоль необходимо ввести “0” на вопрос программы “Строка из консоли или из файла (0/1)?”.

Рекомендуется работать только с консолью, так как она позволяет использовать цвета, что в данной работе активно используется при выводе информации.

Программе подается на вход единственная строка для разбора.

После ввода строки вызывается функция `readLisp(&list)`, которая считывает иерархический список.

После построения иерархического списка запускается функция `parser(lisp s, int depth)`, которая определяет корректность логического выражения.

5. Описание функций:

Все функции описаны в исходном коде в стиле Javadoc.

istream &operator>>(istream &input, Base &b) - Перегруженный оператор ввода атома из строкового потока.

@param input ссылка на поток

@param b объект класса атома

@return ссылка на поток ввода

ostream &operator<<(ostream &out, const Base &b) - Перегруженный оператор вывода атома в строковой поток.

@param out ссылка на поток вывода

@param b объект класса атома

@return ссылка на поток вывода

bool isAtom(lisp s) - Функция проверки иерархического списка на атомарность

@param s список для проверки

@return bool, является ли список атомом.

lisp cons(lisp head, lisp tail) - Функция конструктор иерархического списка.

@param head указатель на логического выражение головы

@param tail указатель на логического выражение хвоста

@return lisp объединенный узел логического выражения

lisp cons(lisp head, lisp tail) - Создает список-атом на основе объекта атома.

@param b - объект атома

@return

void readBase(Base &b, char c) - Вспомогательная функция для удобного ввода-вывода атомов.

@param b объект атома, который формируется в данный момент времени

@param c символ, случайно прочитанный ранее. Необходимо добавить в строковой поток, например, возможна ситуация когда в функции AND символ A прочитан, а ND еще нет.

void readSeq(lisp &list) - Рекурсивная функция чтения иерархического (под)списка

@param list (под)список, который формируется

void readExpr(Base b, char c, lisp &list) - Функция управления чтением. Либо делает (под)список атомом, либо продолжает чтение.

@param b прочитанный атом. Может быть и пуст.

@param с символ на случай если там открывающая или закрывающая скобка

@param list текущий (под)список

void readLisp(lisp &list) - Функция запуска чтения иерархического списка.

@param list список для чтения

lisp head(const lisp s) - Возвращает голову из узла.

@param s логическое выражение(иерархический список)

@return указатель на логическое выражение - голову

lisp tail(const lisp s) - Возвращает хвост из узла

@param s логическое выражение(иерархический список)

@return указатель на логическое выражение - хвост

string write_lisp(const lisp x) - Функция преобразования иерархического списка в строку/

@param x (под)список для преобразования

@return строка, с сформированным (под)списком

string write_seq(const lisp x) - Вспомогательная функция преобразования иерархического списка в строку. Запускает формировку строку из головы и хвоста.

@param x (под)список для преобразования

@return строка, с сформированным (под)списком

void printDepth(lisp s, int depth, bool parsed) - Функция вывода промежуточной информации.

@param - s текущий список для парсинга

@param - depth глубина рекурсии

@param - parsed является ли подстрока корректным списком/атомом

void printError(lisp s, int code, int depth) - Функция вывода ошибки проверки корректности выражения

@param s список с ошибкой

@param code код ошибки

@param depth глубина рекурсии разбора

parserResult parser(lisp s, int depth) - Рекурсивная функция проверки корректности логического выражения. Главным критерием правильности выражения можно обозначить наличие хотя бы одного операнда у каждой функции.

@param s (под)список для проверки

@param depth глубина рекурсии для отладочного вывода

@return parserResult структура с результатом разбора.

6. Описание структур данных

Класс атома. Может представлять как функцию, так и оператор. Операторы ввода-вывода перегружены.

```
class Base { ... }
```

В данной работе для удобства вывода промежуточной информации результат парсинга оборачивается в обертку

```
struct parserResult {  
  
    string s;  
  
    string status;  
  
};
```

где string s - скобка, которую самой последней получил парсер,

string status - результат парсинга скобки

Структура узла, с указателями на голову и хвост

```
struct Node {  
  
    lisp head;  
  
    lisp tail;  
  
};
```

Структура для логического выражения. Одновременно либо узел с другими логическими выражениями, либо атом.

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

8. Вывод:

В ходе выполнения лабораторной работы была создана программа, обрабатывающая логическое выражение, представленное иерархическим списком.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include <bits/stdc++.h>

using namespace std;

#define PARSED "OK"

#define OPERAND "op"

#define FUNCTION "func"

#define LISP "lisp"

#define EMPTY ""

#define FAILURE "ERROR"

#ifdef __linux__

    #define RESET_COLOR    "\033[0m"

    #define FAILURE_COLOR  "\033[1m\033[31m"

    #define SUCCESS_COLOR  "\033[1m\033[32m"

    #define INFO_COLOR     "\033[1m\033[36m"

    #define ATOM_COLOR     "\033[1m\033[33m"

    #define LISP_COLOR     "\033[1m\033[35m"

#elif _WIN32

    #define RESET_COLOR    ""

    #define FAILURE_COLOR  ""

    #define SUCCESS_COLOR  ""

    #define INFO_COLOR     ""

    #define ATOM_COLOR     ""

    #define LISP_COLOR     ""
```

```

#endif

enum Operation {

    AND = 1,

    OR,

    NOT,

    XOR,

    ERROR // нет такой операции
};

/**
 * Класс атома. Может представлять как функцию, так и оператор.
 * Операторы ввода-вывода перегружены.
 */

class Base {
private:

    static bool isStringOperand(const string &s) {

        return s == "true" || s == "false";

    }

    static Operation readFunc(const string &s) {

        if (s == "AND") return AND;

        else if (s == "OR") return OR;

        else if (s == "NOT") return NOT;

        else if (s == "XOR") return XOR;

        else return ERROR;

    }

public:

```

```

bool isOperand; // 1 - op, 0 - func

union {

    bool op;

    Operation func;

} base;

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

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

};

/**
 * Перегруженный оператор ввода атома из строкового потока.
 * @param input ссылка на поток
 * @param b объект класса атома
 * @return ссылка на поток ввода
 */
istream &operator>>(istream &input, Base &b) {
    string s;
    input >> s;
    if (Base::isStringOperand(s)) {
        std::istringstream is(s);
        is >> boolalpha >> b.base.op;
        b.isOperand = true;
    } else {
        b.base.func = Base::readFunc(s);
        b.isOperand = false;
    }
    return input;
}

```

```

/**
 * Перегруженный оператор вывода атома в строковой поток.
 * @param out ссылка на поток вывода
 * @param b объект класса атома
 * @return ссылка на поток вывода
 */
ostream &operator<<(ostream &out, const Base &b) {
    if (b.isOperand) {
        cout << boolalpha << b.base.op;
    } else {
        vector<string> functions = {"", "AND", "OR", "NOT", "XOR"};
        cout << functions[b.base.func];
    }
    return out;
}

struct b_expr;

typedef b_expr *lisp;

/**
 * Структура узла, с указателями на голову и хвост
 */
struct Node {
    lisp head;
    lisp tail;
};

```

```

/**
 * Структура для логического выражения. Одновременно либо узел с другими логическими
 * выражениями, либо атом.
 */
struct b_expr {
    bool isAtom;

    union {
        Base atom;
        Node pair;
    } node;
};

/**
 * Функция проверки иерархического списка на атомарность
 * @param s список для проверки
 * @return bool, является ли список атомом.
 */
bool isAtom(lisp s) {
    if (s == nullptr) return false;
    else return (s->isAtom);
}

/**
 * Функция конструктор иерархического списка.
 * @param head указатель на логического выражение головы
 * @param tail указатель на логического выражение хвоста
 * @return lisp объединенный узел логического выражения
 */
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;

    }

}

/**
 * Создает список-атом на основе объекта атома.
 * @param b - объект атома
 * @return
 */
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);

/**
 * Вспомогательная функция для удобного ввода-вывода атомов.
 * @param b объект атома, который формируется в данный момент времени

```

```

* @param c символ, случайно прочитанный ранее. Необходимо добавить в строковый поток,
* например, возможна ситуация когда в функции AND символ A прочитан, а ND еще нет.
*/

void readBase(Base &b, char c) {
    string s;
    cin >> s;
    if (s.size())
        if (c) {
            istringstream is(c + s);
            is >> b;
        } else {
            istringstream is(s);
            is >> b;
        }
}

/**
* Рекурсивная функция чтения иерархического (под)списка
* @param list (под)список, который формируется
*/

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);
        }

        readExpr(b, c, p1);
        readSeq(p2);
        list = cons(p1, p2);
    }

}

/**
 * Функция управления чтением. Либо делает (под)список атомом, либо продолжает чтение
 * @param b прочитанный атом. Может быть и пуст.
 * @param c символ на случай если там открывающая или закрывающая скобка
 * @param list текущий (под)список
 */
void readExpr(Base b, char c, lisp &list) {
    if (b.base.func == ERROR || c == ')') {
        cerr << " ! List.Error 1 " << endl;
        exit(1);
    } else if (c != '(') list = makeAtom(b);
    else readSeq(list);
}

/**
 * Функция запуска чтения иерархического списка
 * @param list список для чтения
 */
void readLisp(lisp &list) {
    char c;

```

```

do

    cin >> c;

while (c == ' ');

Base b;

if (c != '(' && c != ')') {

    readBase(b, c);

}

readExpr(b, c, list);

}

/**
 * Возвращает голову из узла
 * @param s логическое выражение (иерархический список)
 * @return указатель на логическое выражение - голову
 */
lisp head(const lisp s) {

    if (s != nullptr)

        if (!isAtom(s)) return s->node.pair.head;

        else {

            cerr << "Error: Head(atom) \n";

            exit(1);

        }

    else {

        cerr << "Error: Head(nil) \n";

        exit(1);

    }

}

/**
 * Возвращает хвост из узла

```

```

* @param s логическое выражение (иерархический список)
* @return указатель на логическое выражение - хвост
*/

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

string write_seq(lisp x);

/**
* Функция преобразования иерархического списка в строку
* @param x (под)список для преобразования
* @return строка, с сформированным (под)списком
*/

string write_lisp(const lisp x) {
    string s;

    if (x == nullptr) s = "( )";

    else if (isAtom(x)) {
        if (x->node.atom.isOperand) {
            s = x->node.atom.base.op ? "true " : "false ";
        } else {

```

```

        vector<string> functions = {"", "AND ", "OR ", "NOT ", "XOR "};

        s = functions[x->node.atom.base.func];

    }

} else {

    s += "( ";

    s += write_seq(x);

    s += ") ";

}

return s;

}

```

/**

* Вспомогательная функция преобразования иерархического списка в строку.

* Запускает формирование строки из головы и хвоста.

* @param x (под)список для преобразования

* @return строка, с сформированным (под)списком

*/

```

string write_seq(const lisp x) {

    string s;

    if (x != nullptr) {

        s += write_lisp(x->node.pair.head);

        s += write_seq(x->node.pair.tail);

    }

    return s;

}

```

/**

* Структура-обертка результата парсинга

* Помещается строка s - самое последнее, что обнаружил парсер и

* статус - успешен ли парсинг.

*/

```

struct parserResult {
    string status;
    string s;
};

/**
 * Функция вывода промежуточной информации.
 * @param - s текущий список для парсинга
 * @param - depth глубина рекурсии
 * @param - parsed является ли подстрока корректным списком/атомом
 */
void printDepth(lisp s, int depth, bool parsed) {
    cout << RESET_COLOR;
    for (int i = 0; i < depth; i++) cout << "|\\t"; // табуируемся по глубине рекурсии
    if (!parsed) {
        cout << INFO_COLOR << "Parser for " << LISP_COLOR << write_lisp(s) << INFO_COLOR
" is called\\n";
    } else {
        if (isAtom(s)) {
            if (s->node.atom.isOperand) {
                cout << INFO_COLOR << write_lisp(s) << ATOM_COLOR << " is correct
operand\\n";
            } else {
                cout << INFO_COLOR << write_lisp(s) << ATOM_COLOR << " is correct
function\\n";
            }
        } else {
            cout << INFO_COLOR << write_lisp(s) << SUCCESS_COLOR << " is correct lisp\\n";
        }
    }
    cout << RESET_COLOR;
}

```

```

/**
 * Функция вывода ошибки проверки корректности выражения
 * @param s список с ошибкой
 * @param code код ошибки
 * @param depth глубина рекурсии разбора
 */
void printError(lisp s, int code, int depth) {
    for (int i = 0; i < depth; i++) cout << "|\t"; // табуируемся по глубине рекурсии
    cout << FAILURE_COLOR << FAILURE;

    switch (code) {
        case 1:
            cout << ": didn't found operand after function";
            break;
        case 2:
            cout << ": cannot parse head";
            break;
        case 3:
            cout << ": didn't found operand after function";
            break;
        case 4:
            cout << ": couldn't parse both head and tail";
            break;
    }

    cout << " for " << LISP_COLOR << write_lisp(s) << "\n" << RESET_COLOR;
}

/**
 * Рекурсивная функция проверки корректности логического выражения.

```



```

* Главным критерием правильности выражения можно обозначить наличие хотя бы одного
* операнда у каждой функции.
* @param s (под)список для проверки
* @param depth глубина рекурсии для отладочного вывода
* @return parserResult структура с результатом разбора.
*/

parserResult parser(lisp s, int depth) {
    lisp h, t;
    parserResult res, headRes, tailRes;

    printDepth(s, depth, false);

    if (s == nullptr) {
        printDepth(s, depth, true);
        return {PARSED, EMPTY};
    } else if (isAtom(s)) {
        printDepth(s, depth, true);
        if (s->node.atom.isOperand) return {PARSED, OPERAND};
        else return {PARSED, FUNCTION};
    } else {
        h = head(s);
        t = tail(s);

        if (t == nullptr && h != nullptr) {
            res = parser(h, depth + 1);
            if (res.status == PARSED) {
                if (res.s == FUNCTION) {
                    printError(s, 1, depth);
                    return {FAILURE, ""};
                } else {

```

```

        printDepth(s, depth, true);

        return {PARSED, OPERAND};

    }

    } else {

        printError(s, 2, depth);

        return {FAILURE, ""};

    }

} else if (t == nullptr && h == nullptr) {

    printDepth(s, depth, true);

    return {PARSED, EMPTY};

} else {

    headRes = parser(head(s), depth + 1);

    tailRes = parser(tail(s), depth + 1);

    if (headRes.status == PARSED && tailRes.status == PARSED) {

        if (headRes.s == FUNCTION) {

            if (tailRes.s == OPERAND || tailRes.s == LISP) {

                printDepth(s, depth, true);

                return {PARSED, ""};

            } else {

                printError(s, 3, depth);

                return {FAILURE, ""};

            }

        } else if (headRes.s == OPERAND) {

            printDepth(s, depth, true);

            return {PARSED, LISP};

        }

    } else {

        printError(s, 4, depth);

        return {FAILURE, ""};

    }

}

```

```

    }

}

return {PARSED, ""};

}

int main() {

    int f;

    cout << "Строка из консоли или из файла (0/1)?\n";

    cin >> f;

    if (f == 1) {

        freopen("input.txt", "r", stdin);

        //Если работать на windows, то можно использовать вывод в файл. Если нет, то в
        файл

        //будет выводить абракадабра с цветами.

        //      freopen("output.txt", "w", stdout);

    }

    else cout << "Введите строку :\n";

    lisp l;

    readLisp(l);

    cout << "Read : " << write_lisp(l);

    cout << "\n";

    parserResult res = parser(l, 0);

    cout << INFO_COLOR << "\n\n\n Lisp ";

    if (res.status == PARSED) {

        cout << SUCCESS_COLOR << "correct";

    } else {

        cout << FAILURE_COLOR << "not correct";

    }

    return 0;

}

```

