**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЯ ОБРАЗОВАНИЯ**

**“ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ ЕВФРОСИНИИ ПОЛОЦКОЙ”**

Факультет информационных технологий

Кафедра технологий программирования

**Отчёт по Лабораторной работе №3**

**«Построение простейшего дерева вывода»**

Выполнил студент 2 курса, группа 21-ИТ-1 Макеёнок Д.И.

Проверил Сыцевич Д.Н.

Полоцк, 2022 г.

**Цель работы:** изучение основных понятий теории грамматик простого и

операторного предшествования, ознакомление с алгоритмами

синтаксического анализа (разбора) для некоторых классов КС-грамматик,

получение практических навыков создания простейшего синтаксического

анализатора для заданной грамматики операторного предшествования.

**Результат работы:**

#include<iostream>

#include<string>

#include<list>

using namespace std;

enum tok\_names { semocolon, ident, num, asgn, parentheses, OR, boolean, constants };

struct token

{

enum tok\_names token\_name;

string token\_names[11] = { "Semocolon", "ident", "Number", "Assign", "Parentheses", "OR", "boolean", "constants" };

string token\_value;

};

list<token> lexeme\_table;

token add\_token(tok\_names a, string b) {

token tok;

tok.token\_name = a;

tok.token\_value = b;

return tok;

}

list<token> lexer(string str)

{

list<token> lexeme\_table;

int i = 0; int value = 0;

while (i < str.size())

{

if (str[i] == ':' and str[i + 1] == '=') {

lexeme\_table.push\_back(add\_token(asgn, ":="));

i += 2;

}

if (str[i] == ';') {

lexeme\_table.push\_back(add\_token(semocolon, ";"));

}

if (str[i] == '(') {

lexeme\_table.push\_back(add\_token(parentheses, "("));

}

if (str[i] == ')') {

lexeme\_table.push\_back(add\_token(parentheses, ")"));

}

if (str[i] == 'o' and str[i + 1] == 'r') {

lexeme\_table.push\_back(add\_token(OR, "or"));

i += 2;

}

if (str[i] == '<') {

string con = "<";

i++;

while (str[i] >= '0' and str[i] <= '9')

{

con += str[i];

i++;

}

con += ">";

lexeme\_table.push\_back(add\_token(constants, con));

}

if (str[i] == '|') {

lexeme\_table.push\_back(add\_token(OR, "or"));

i++;

}

if ((str[i] >= 'a' and str[i] <= 'z') or (str[i] >= 'A' and str[i] <= 'Z')) {

string var = "";

while ((str[i] >= 'a' and str[i] <= 'z') or (str[i] >= 'A' and str[i] <= 'Z') or (str[i] >= '0' and str[i] <= '9'))

{

var += str[i];

i++;

}

i--;

lexeme\_table.push\_back(add\_token(ident, var));

}

if ((str[i] >= '0' and str[i] <= '9')) {

string number = "";

while (str[i] >= '0' and str[i] <= '9')

{

number += str[i];

i++;

}

i--;

lexeme\_table.push\_back(add\_token(num, number));

}

i++;

}

return lexeme\_table;

}

struct node

{

token token; //Информационное поле

node\* l, \* r; //Левая и Правая часть дерева

};

node\* tree = NULL; //Объявляем переменную, тип которой структура Дерево

void pushTree(token token, node\*\* t) //Add

{

if ((\*t) == NULL) //Если дерева не существует

{

(\*t) = new node; //Выделяем память

(\*t)->token = token; //Кладем в выделенное место аргумент a

(\*t)->l = (\*t)->r = NULL; //Очищаем память для следующего роста

return; //Заложили семечко, выходим

}

if (token.token\_name != constants and token.token\_name != num and token.token\_name != ident) {

pushTree(token, &(\*t)->r);

return;

}

if ((token.token\_name == constants or token.token\_name == num or token.token\_name == ident) and (\*t)->l == NULL) {

pushTree(token, &(\*t)->l);

return;

}

else {

pushTree(token, &(\*t)->r);

return;

}

}

void printTree(node\* t, int u, bool Direction)

{

if (t == NULL) return; //Если дерево пустое, то отображать нечего, выходим

else

{

printTree(t->l, ++u, 1); //С помощью рекурсивного посещаем левое поддерево

cout << u;

cout << ' ';

if (u != 1) Direction == 1 ? cout << "l " : cout << "r ";

cout << t->token.token\_value << endl;

u--;

}

printTree(t->r, ++u, 0); //С помощью рекурсии посещаем правое поддерево

}

int main()

{

token tok;

string str = "fortnite or PABAG ; <9678>";

list<token> lexeme\_table = lexer(str);

list<token> temp = lexeme\_table;

while (temp.empty() == 0) {

tok = temp.front();

cout << "name:" << tok.token\_names[tok.token\_name] << ", " << "value:" << tok.token\_value << endl;

temp.pop\_front();

}

cout << endl;

list<token> znak;

list<token> identif;

temp = lexeme\_table;

while (temp.size() > 0)

{

if (temp.front().token\_name == num or temp.front().token\_name == constants or temp.front().token\_name == ident) {

identif.push\_back(temp.front());

temp.pop\_front();

}

else {

znak.push\_back(temp.front());

temp.pop\_front();

}

}

node\* sosna = NULL;

while (znak.size() > 0 or identif.size() > 0)

{

if (znak.size() > 0) {

pushTree(znak.front(), &sosna);

znak.pop\_front();

}

else

{

pushTree(identif.front(), &sosna);

identif.pop\_front();

}

}

printTree(sosna, 0, 1);

}

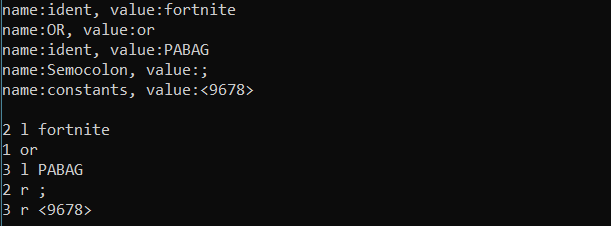


Рисунок 1 – Результат работы программы

**Вывод:** В результате выполнения написанного программного кода было построено бинарное дерево и создан простейший синтаксический

анализатор для заданной грамматики операторного предшествования.