Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа № 4

по дисциплине: «Конструирование компиляторов»

Тема: «Синтаксический анализатор операторного предшествования»

Выполнил студент группы ИУ7-21М

Констандогло Александр

Москва
15 мая 2020

Цель работы: приобретение практических навыков реализации таблично управляемых синтаксических анализаторов на примере анализатора операторного предшествования.

Задачи работы:

1. Ознакомиться с основными понятиями и определениями, лежащими в основе синтаксического анализа операторного предшествования.
2. Изучить алгоритм синтаксического анализа операторного предшествования.
3. Разработать, тестировать и отладить программу синтаксического анализа в соответствии с предложенным вариантом грамматики.
4. Включить в программу синтаксического анализ семантические действия для реализации синтаксически управляемого перевода инфиксного выражения в обратную польскую нотацию.

Постановка задачи

Вариант 7.

Реализовать синтаксический анализатор операторного предшествования и синтаксически управляемый перевод инфиксного выражения в обратную польскую нотацию для грамматики **G**.

Грамматика G

```
<выражение> ->
     <арифметическое выражение> <операция отношения> <арифметическое
     выражение> |
     <арифметическое выражение>
<арифметическое выражение> ->
     <арифметическое выражение> <операция типа сложения> <терм> |
     <терм>
<терм> ->
     <терм> <операция типа умножения> <фактор> |
     <фактор>
<фактор> ->
     identifier |
     const |
     ( <арифметическое выражение> )
<операция отношения> ->
     < | <= | = | <> | > | >=
<операция типа сложения> ->
     + | -
<операция типа умножения> ->
     * | /
```

Замечания.

1. Терминалы ( ) – это разделители и символы пунктуации.

2. Терминалы < <= = <> > >= + - * / – это знаки операций.
3. Нетерминал <выражение> – это начальный символ грамматики.

Матрица отношений операторного предшествования для грамматики G

	identifier, const	+, -	*, /	<, <=, =, <>, >, >=	(	)	$
identifier, const		>	>	>		>	>
+, -	<	>	<	>	<	>	>
*, /	<	>	>	>	<	>	>
<, <=, =, <>, >, >=	<	<	<		<		>
(	<	<	<	<	<	=	
)		>	>	>		>	>
$	<	<	<	<	<		

Остовная грамматика для G

```
<выражение> ->
      <выражение> <= <выражение> |
      <выражение> < <выражение> |
      <выражение> <> <выражение> |
      <выражение> >= <выражение> |
      <выражение> > <выражение> |
      <выражение> = <выражение> |
      <выражение> + <выражение> |
      <выражение> - <выражение> |
      <выражение> * <выражение> |
      <выражение> / <выражение> |
      ( <выражение> ) |
      identifier |
      const
```

Текст программы, проводящей нисходящий синтаксический анализ

Программа написана на языке C++. Для ее корректной работы в операционной системе должна быть установлена утилита dot.

parser.h

```
#pragma once
#include <map>
#include <set>
#include <vector>
#include <string>

enum Terminal {
        ATOM, SUM, MULT, LOGIC, LP, RP, END
};

class Node {
        int num;
        std::string name;
        bool terminal;
        std::vector<Node*> children;
public:
        void set_childcnt(int cnt);
        Node(int num, std::string name, bool terminal);
        void add_child(Node *child, int num);
```

```
        int get_num();
        std::string get_name();
        bool get_terminal();
        std::vector<Node*> get_children();
        ~Node();
};

class Parser {
        // Матрица отношений операторного предшествования
        const std::vector<std::vector<char>> matrix = std::vector<std::vector<char>>({
                //          |                                                  atm  sum  mul  log  lp   rp   end
                // ----+-------------------------------------------------------
                /* atm | */ std::vector<char>({'1', '>', '>', '>', '4', '>', '>'}),
                /* sum | */ std::vector<char>({'<', '>', '<', '>', '<', '>', '>'}),
                /* mul | */ std::vector<char>({'<', '>', '>', '>', '<', '>', '>'}),
                /* log | */ std::vector<char>({'<', '<', '<', '2', '<', '3', '>'}),
                /* lp  | */ std::vector<char>({'<', '<', '<', '<', '<', '=', '6'}),
                /* rp  | */ std::vector<char>({'5', '>', '>', '>', '7', '>', '>'}),
                /* end | */ std::vector<char>({'<', '<', '<', '<', '<', '8', '0'}) });
        const std::string marker = "$";
        std::map<std::string, Terminal> type_of_token = std::map<std::string, Terminal>({
                        std::pair<std::string, Terminal>("identifier", ATOM),
                        std::pair<std::string, Terminal>("const", ATOM),
                        std::pair<std::string, Terminal>("<=", LOGIC),
                        std::pair<std::string, Terminal>("<>", LOGIC),
                        std::pair<std::string, Terminal>("<", LOGIC),
                        std::pair<std::string, Terminal>(">=", LOGIC),
                        std::pair<std::string, Terminal>(">", LOGIC),
                        std::pair<std::string, Terminal>("=", LOGIC),
                        std::pair<std::string, Terminal>("+", SUM),
                        std::pair<std::string, Terminal>("-", SUM),
                        std::pair<std::string, Terminal>("*", MULT),
                        std::pair<std::string, Terminal>("/", MULT),
                        std::pair<std::string, Terminal>("(", LP),
                        std::pair<std::string, Terminal>(")", RP),
                        std::pair<std::string, Terminal>(marker, END) });
        const std::vector<std::string> terminals = std::vector<std::string>({
                "identifier", "const", "<=", "<>", "<", ">=", ">", "=", "+", "-", "*", "/", "(", ")", marker});
        // Входная строка
        std::string text;
        // Стек разбора
        std::vector<std::string> st;
        bool prnts;
        bool oper;
        std::vector<Node*> tree;
        std::vector<std::string> postf;
        std::pair<bool, std::string> current;
        std::vector<std::string> symbols;
        void make_graph();
        void print_sate(bool reduct);
        void gen_postf(std::string sym);
        std::pair<bool, std::string> next_terminal();
        void make_tree();
public:
        Parser(std::string txt);
        bool parse();
        void handle();
};
```

parser.cpp

```
#include "parser.h"
#include <algorithm>
#include <fstream>
#include <iostream>
#include <regex>
#include <stack>

using namespace std;

Node::Node(int num, string name, bool terminal) : num(num), name(name), terminal(terminal)
{ }

void Node::set_childcnt(int cnt)
{
        children = vector<Node*>(cnt);
}

void Node::add_child(Node *child, int num)
{
        children[num] = child;
}

int Node::get_num()
{
```

```
	return num;
}

string Node::get_name()
{
	return name;
}

bool Node::get_terminal()
{
	return terminal;
}

vector<Node*> Node::get_children()
{
	return children;
}

Node::~Node()
{
	children.clear();
}

pair<bool, string> Parser::next_terminal()
{
	if (text.size() == 0)
		return pair<bool, string>(false, "");
	string ret;
	for (string s : terminals)
	{
		if (s.size() > text.size())
			continue;
		else if (s == text.substr(0, s.size()))
		{
			text = text.substr(s.size());
			prnts = type_of_token[s] == RP;
			oper = (type_of_token[s] == LOGIC || type_of_token[s] == SUM || type_of_token[s] == MULT);
			return pair<bool, string>(true, s);
		}
	}
	ret.push_back(text[0]);
	text = text.substr(1);
	return pair<bool, string>(false, ret);
}

void Parser::gen_postf(string sym)
{
	if (type_of_token[sym] != LP && type_of_token[sym] != RP)
		postf.push_back(sym);
}

void Parser::print_sate(bool reduct)
{
	cout << endl << ((reduct) ? "r" : "s") << " [";
	for (string x : st)
		cout << x << " ";
	cout << ",\t" <<current.second << text << "] |-";
}

bool Parser::parse()
{
	string tmp = text;
	bool ans = true;
	st.push_back(marker);
	current = next_terminal();
	print_sate(false);
	while (current.second != "")
	{
		if (!current.first)
		{
			cout << endl << "ОШИБКА: Обнаружен недопустимый символ " << "\"" << current.second << "\"." << endl;
			ans = false;
			current = next_terminal();
			continue;
		}
		switch (matrix[type_of_token[st[st.size()-1]]][type_of_token[current.second]])
		{
		case '<':
```

```
                case '=': // Перенос
                        if (prnts && type_of_token[st[st.size() - 1]] == LP)
                        {
                                cout << endl << "ОШИБКА: Между открывающей и закрывающей скобками отсутствует арифметическое
выражение" << endl;
                                ans = false;
                        }
                        if (oper && (type_of_token[st[st.size() - 1]] == LOGIC || type_of_token[st[st.size() - 1]] == SUM ||
type_of_token[st[st.size() - 1]] == MULT))
                        {
                                cout << endl << "ОШИБКА: Бинарные операторы записаны подряд" << endl;
                                ans = false;
                        }
                        st.push_back(current.second);
                        current = next_terminal();
                        print_sate(false);
                        break;
                case '>': // Свёртка
                        prnts = false;
                        if (oper && (type_of_token[st[st.size() - 1]] == LOGIC || type_of_token[st[st.size() - 1]] == SUM ||
type_of_token[st[st.size() - 1]] == MULT))
                        {
                                cout << endl << "ОШИБКА: Бинарные операторы записаны подряд" << endl;
                                ans = false;
                        }
                        do {
                                if (!(type_of_token[st[st.size() - 1]] == LOGIC || type_of_token[st[st.size() - 1]] == SUM
|| type_of_token[st[st.size() - 1]] == MULT))
                                        oper = false;
                                postf.push_back(st[st.size() - 1]);
                                st.pop_back();
                                print_sate(true);
                        } while (postf.size()>0 && matrix[type_of_token[st[st.size() - 1]]][type_of_token[postf[postf.size() -
1]]] != '<');
                        break;
                case '0':
                        if (tmp.size() == 1) {
                                cout << endl << "ОШИБКА: Строка пуста" << endl;
                                ans = false;
                        }
                        else
                                cout << endl << "Допуск" << endl;
                        current = next_terminal();

                        break;
                case '1':
                        cout << endl << "ОШИБКА: Между факторами отсутствует оператор" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                case '2':
                        cout << endl << "ОШИБКА: В выражении больше одного логического оператора" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                case '3':
                        cout << endl << "ОШИБКА: После логического оператора недостаёт открывающей скобки" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                case '4':
                        cout << endl << "ОШИБКА: После арифметического выражения перед открывающей скобкой отсутствует
бинарный оператор" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                case '5':
                        cout << endl << "ОШИБКА: После закрывающей скобки перед арифметическим выражением отсутствует бинарный
оператор" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                case '6':
                        cout << endl << "ОШИБКА: Отсутствует закрывающая скобка" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                case '7':
```

```
                        cout << endl << "ОШИБКА: Между закрывающей и открывающей скобками отсутствует бинарный оператор" <<
endl;
                        current = next_terminal();
                        ans = false;
                        break;
                case '8':
                        cout << endl << "ОШИБКА: Отсутствует открывающая" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                default:
                        cout << endl << "ОШИБКА: Тип ошибки неизвестен" << endl;
                        current = next_terminal();
                        ans = false;
                        break;
                }
        }
        text = tmp;
        return ans;
}

Parser::Parser(string txt)
{
        text = txt + marker;
        prnts = false;
        oper = false;
}

void Parser::make_tree()
{
        const string E = "E";
        int num = 0;
        stack<Node*> roots;
        tree.push_back(new Node(num++, E, false));
        roots.push(tree[0]);
        for (int i = postf.size() - 1; i > -1; i--)
        {
                if (type_of_token[postf[i]] == LP) // (E)
                {
                        roots.top()->set_childcnt(3);
                        tree.push_back(new Node(num++, "(", true));
                        roots.top()->add_child(tree[tree.size() - 1], 0);
                        tree.push_back(new Node(num++, E, false));
                        roots.top()->add_child(tree[tree.size() - 1], 1);
                        tree.push_back(new Node(num++, ")", true));
                        roots.top()->add_child(tree[tree.size() - 1], 2);
                        i--;
                        roots.pop();
                        roots.push(tree[tree.size() - 2]);
                }
                else if (type_of_token[postf[i]] == LOGIC || type_of_token[postf[i]] == SUM || type_of_token[postf[i]] == MULT)
// E<operator>E
                {
                        roots.top()->set_childcnt(3);
                        tree.push_back(new Node(num++, E, false));
                        roots.top()->add_child(tree[tree.size() - 1], 0);
                        tree.push_back(new Node(num++, postf[i], true));
                        roots.top()->add_child(tree[tree.size() - 1], 1);
                        tree.push_back(new Node(num++, E, false));
                        roots.top()->add_child(tree[tree.size() - 1], 2);
                        roots.pop();
                        roots.push(tree[tree.size() - 3]);
                        roots.push(tree[tree.size() - 1]);
                }
                else if (type_of_token[postf[i]] == ATOM) // <atom>
                {
                        roots.top()->set_childcnt(1);
                        tree.push_back(new Node(num++, postf[i], true));
                        roots.top()->add_child(tree[tree.size() - 1], 0);
                        roots.pop();
                }
        }
}

void Parser::make_graph()
{
        ofstream tree_dot("tree.dot");
        if (!tree_dot.is_open())
```

```
                throw runtime_error("Не удалось открыть файл tree.dot для записи\n");
        tree_dot << "digraph G{" << endl << "node[shape=rectangle style=filled fillcolor=white fontsize=12];" << endl;
        for (Node* node : tree)
        {
                tree_dot << node->get_num() << "[label=\"" << node->get_name() << "\"";
                if (node->get_terminal())
                        tree_dot << " fillcolor=lightgrey shape=ellipse";
                tree_dot << "];" << endl;
                for (Node* child : node->get_children())
                        tree_dot << node->get_num() << "->" << child->get_num() << ";" << endl;
        }
        tree_dot << "}" << endl;
        tree_dot.close();
        system("dot -Tsvg tree.dot -o tree.svg");
}

void Parser::handle()
{
        bool ok = parse();
        cout << endl;
        if (ok)
        {
                cout << "Постфиксная запись:" << endl;
                for (string x : postf)
                        if (type_of_token[x] != LP && type_of_token[x] != RP)
                        cout << x << " ";
                cout << endl;
                make_tree();
                make_graph();
        }
}
```

Source.cpp

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

using namespace std;

int main(int argc, char* argv[])
{
        setlocale(LC_ALL, "Russian");
        cout << "Введите выражение:" << endl;
        string str;
        getline(cin, str);
        try
        {
                Parser p(str);
                p.handle();
        }
        catch (const exception& err) {
                cerr << err.what() << endl;
        }
        return 0;
}
```

Результаты тестирования

Файл `UnitTest.cpp` содержит тесты для проверки выражений. Далее приведены тесты.

UnitTest.cpp

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../parser/parser.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest
{
        TEST_CLASS(UnitTest)
        {
        public:

                TEST_METHOD(Test1)
                {
                        Assert::IsTrue(Parser("const+(const*(const+const))/identifier>=const*(identifier)").parse());
                }

                TEST_METHOD(Test2)
```

```
		{
			Assert::IsTrue(Parser("const-const").parse());
		}

		TEST_METHOD(Test3)
		{
			Assert::IsTrue(Parser("const").parse());
		}

		TEST_METHOD(Test4)
		{
			Assert::IsTrue(Parser("((const))").parse());
		}

		TEST_METHOD(Test5)
		{
			Assert::IsFalse(Parser("").parse());
		}

		TEST_METHOD(Test6)
		{
			Assert::IsFalse(Parser("const-+const").parse());
		}

		TEST_METHOD(Test7)
		{
			Assert::IsFalse(Parser("const<const/identifier>=const*identifier").parse());
		}

		TEST_METHOD(Test8)
		{
			Assert::IsFalse(Parser("(const<=const*identifier)").parse());
		}

		TEST_METHOD(Test9)
		{
			Assert::IsFalse(Parser("(const").parse());
		}

		TEST_METHOD(Test10)
		{
			Assert::IsFalse(Parser("const)").parse());
		}

		TEST_METHOD(Test11)
		{
			Assert::IsFalse(Parser("()").parse());
		}
	};
}
```

№	Название функции	Входная строка	Наличие ошибок	Результат теста
1	Test1	const+(const*(const+const))/identifier>=const*(identifier)	Нет	Пройден
2	Test2	const-const	Нет	Пройден
3	Test3	const	Нет	Пройден
4	Test4	((const))	Нет	Пройден
5	Test5		Есть	Пройден
6	Test6	const-+const	Есть	Пройден
7	Test7	const<const/identifier>=const*identifier	Есть	Пройден
8	Test8	(const<=const*identifier)	Есть	Пройден
9	Test9	(const	Есть	Пройден
10	Test10	const)	Есть	Пройден
11	Test11	()	Есть	Пройден

Результаты выполнения программы

Программа принимает строку. Результатом является множество строк, описывающих работу синтаксического анализатора. Входная строка является корректным выражением

тогда и только тогда, когда в процессе разбора не возникает сообщений об ошибках. Если входная строка является корректным выражением, то выводится ещё и постфиксная запись этого выражения, а также создаётся файл в формате SVG с визуализацией дерева разбора выражения. Все нетерминалы в дереве обозначены прямоугольниками, терминалы – овалами.

Ниже приведены примеры выполнения программы.

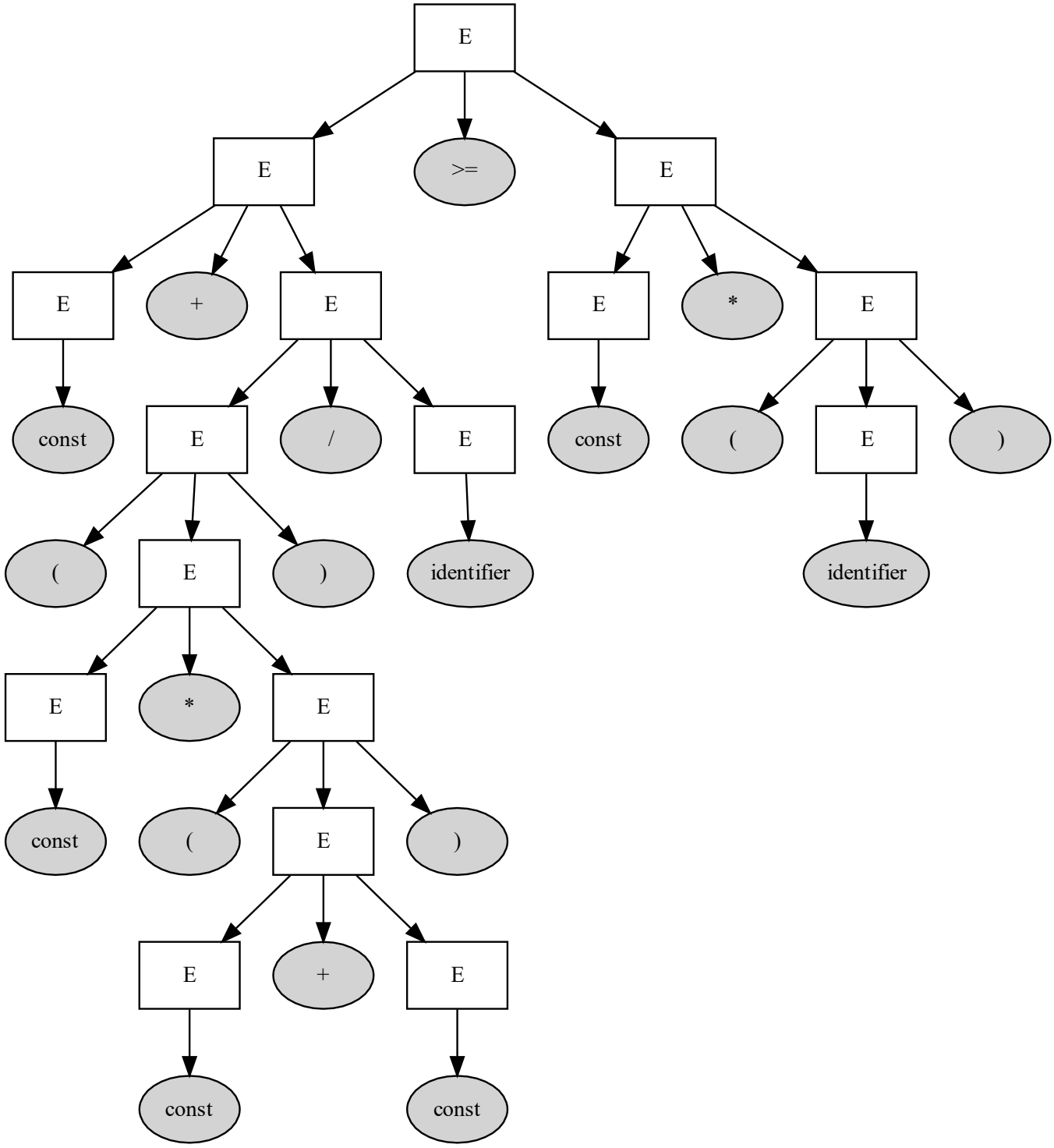
Пример 1.

```
Введите выражение:
const+(const*(const+const))/identifier>=const*(identifier)

s [$ ,  const+(const*(const+const))/identifier>=const*(identifier)$] |-
s [$ const ,    +(const*(const+const))/identifier>=const*(identifier)$] |-
r [$ ,  +(const*(const+const))/identifier>=const*(identifier)$] |-
s [$ + ,        (const*(const+const))/identifier>=const*(identifier)$] |-
s [$ + ( ,      const*(const+const))/identifier>=const*(identifier)$] |-
s [$ + ( const ,        *(const+const))/identifier>=const*(identifier)$] |-
r [$ + ( ,      *(const+const))/identifier>=const*(identifier)$] |-
s [$ + ( * ,    (const+const))/identifier>=const*(identifier)$] |-
s [$ + ( * ( ,  const+const))/identifier>=const*(identifier)$] |-
s [$ + ( * ( const ,    +const))/identifier>=const*(identifier)$] |-
r [$ + ( * ( ,  +const))/identifier>=const*(identifier)$] |-
s [$ + ( * ( + ,        const))/identifier>=const*(identifier)$] |-
s [$ + ( * ( + const ,  ))/identifier>=const*(identifier)$] |-
r [$ + ( * ( + ,        ))/identifier>=const*(identifier)$] |-
r [$ + ( * ( ,  ))/identifier>=const*(identifier)$] |-
s [$ + ( * ( ) ,        )/identifier>=const*(identifier)$] |-
r [$ + ( * ( ,  )/identifier>=const*(identifier)$] |-
r [$ + ( * ,    )/identifier>=const*(identifier)$] |-
r [$ + ( ,      )/identifier>=const*(identifier)$] |-
s [$ + ( ) ,    /identifier>=const*(identifier)$] |-
r [$ + ( ,      /identifier>=const*(identifier)$] |-
r [$ + ,        /identifier>=const*(identifier)$] |-
s [$ + / ,      identifier>=const*(identifier)$] |-
s [$ + / identifier ,   >=const*(identifier)$] |-
r [$ + / ,      >=const*(identifier)$] |-
r [$ + ,        >=const*(identifier)$] |-
r [$ ,  >=const*(identifier)$] |-
s [$ >= ,       const*(identifier)$] |-
s [$ >= const , *(identifier)$] |-
r [$ >= ,       *(identifier)$] |-
s [$ >= * ,     (identifier)$] |-
s [$ >= * ( ,   identifier)$] |-
s [$ >= * ( identifier ,        )$] |-
r [$ >= * ( ,   )$] |-
s [$ >= * ( ) , $] |-
r [$ >= * ( ,   $] |-
r [$ >= * ,     $] |-
r [$ >= ,       $] |-
r [$ ,  $] |-
```

```
Допуск

Постфиксная запись:
const const const const + * identifier / + const identifier * >=
```



Пример 2.

```
Введите выражение:
()++cconst

s [$ ,  ()++cconst$] |-
s [$ ( ,        )++cconst$] |-
```

```
ОШИБКА: Между открывающей и закрывающей скобками отсутствует арифметическое
выражение

s [$ ( ) ,      ++cconst$] |-
r [$ ( ,        ++cconst$] |-
r [$ ,  ++cconst$] |-
s [$ + ,        +cconst$] |-
ОШИБКА: Бинарные операторы записаны подряд

r [$ ,  +cconst$] |-
s [$ + ,        cconst$] |-
ОШИБКА: Обнаружен недопустимый символ "c".

s [$ + const ,  $] |-
r [$ + ,        $] |-
r [$ ,  $] |-
Допуск
```

Выводы

В результате выполнения лабораторной работы был реализован синтаксический анализатор операторного предшествования и перевод инфиксного выражения в обратную польскую нотацию. Также проведена визуализация дерева разбора.