Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Факультет Программной инженерии и компьютерной техники

**Отчет о проделанной работе**

по дисциплине

«Основы разработки компиляторов»

**Выполнили**

Ореховский А.,

Рафиков М.

группа P3317

**Преподаватель**

Лаздин А. В.

Санкт-Петербург

2020

# Исходная грамматика

Вариант 10

<Программа> ::= <Объявление переменных> <Описание вычислений> .

<Описание вычислений> ::= <Список операторов>

<Объявление переменных> ::= Var <Список переменных>

<Список переменных> ::= <Идент> | <Идент> , <Список переменных>

<Список операторов> ::= <Оператор> | <Оператор> <Список операторов>

<Оператор>::=<Присваивание> |<Сложный оператор>

<Присваивание> ::= <Идент> = <Выражение>

<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>

<Подвыражение> :: = ( <Выражение> ) | <Операнд> | <Подвыражение > <Бин.оп.> <Подвыражение>

<Ун.оп.> ::= "-"|"not"

<Бин.оп.> ::= "-" | "+" | "\*" | "/" |"<"|">"|"=="

<Операнд> ::= <Идент> | <Const>

<Сложный оператор>:: =<Оператор цикла>|<Составной оператор>

<Оператор цикла>:: =WHILE <Выражение> DO <Оператор>

<Составной оператор>::= Begin <Список операторов> End

<Идент> ::= <Буква> <Идент> | <Буква>

<Const> ::= <Цифра> <Const> | <Цифра>

# Пример программы на «Языке программирования»

Var a, variable

a = -5

variable = 1

WHILE a < 20 DO

Begin

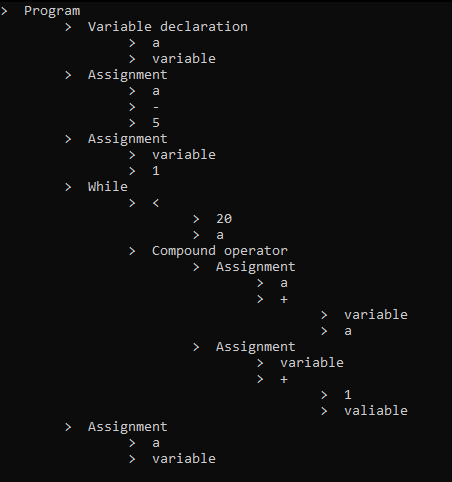
a = a + variable

variable = variable + 1

End

a = variable.

# Абстрактное синтаксическое дерево



# Грамматика лексера

namespace SyntaxAnalysisLibray.Lexer

{

static class Grammar

{

private static readonly Dictionary<string, string> s\_rules = new Dictionary<string, string>();

public static Dictionary<TokenType, string> TokenDefinitions { get; } = new Dictionary<TokenType, string>();

static Grammar()

{

// Правила грамматики

s\_rules.Add("Comma", $"(,)");

s\_rules.Add("Space", $"( )");

s\_rules.Add("LeftBracket", $"(\\()");

s\_rules.Add("RightBracket", $"(\\))");

s\_rules.Add("Digit", $"(0|1|2|3|4|5|6|7|8|9)");

s\_rules.Add("Letter", $"(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)");

s\_rules.Add("Const", $"({s\_rules["Digit"]}+)");

s\_rules.Add("Ident", $"({s\_rules["Letter"]}({s\_rules["Letter"]}|{s\_rules["Digit"]})\*)");

s\_rules.Add("Var", $"(Var)");

s\_rules.Add("EqualSign", $"(=)");

s\_rules.Add("UnaryOperator", $"(not|-)");

s\_rules.Add("BinaryOperator", $"(-|\\+|\\\*|\\/|<|>|==)");

s\_rules.Add("While", $"(WHILE)");

s\_rules.Add("Do", $"(DO)");

s\_rules.Add("Begin", $"(Begin)");

s\_rules.Add("End", $"(End)");

s\_rules.Add("LineBreak", $"(\r\n)");

s\_rules.Add("Tab", $"(\t)");

s\_rules.Add("Dot", $"(\\.)");

// Соответствие между правилами и определениями токенов

foreach (TokenType tokenType in Enum.GetValues(typeof(TokenType)))

{

if (tokenType != TokenType.EOF)

{

TokenDefinitions.Add(tokenType, "^" + s\_rules[UniformEnumToString(tokenType, Capitalization.AsListed)]);

}

}

}

}

}

# Грамматика парсера

namespace SyntaxAnalysisLibray.Parser

{

static class Grammar

{

public static Dictionary<NonTerminal, List<List<object>>> Rules { get; } = new Dictionary<NonTerminal, List<List<object>>>();

static Grammar()

{

Rules.Add(NonTerminal.Root, new List<List<object>>

{

new List<object>{NonTerminal.VariableDeclaration, NonTerminal.ComputingDescription, Terminal.Dot, Terminal.EOF}

});

Rules.Add(NonTerminal.ComputingDescription, new List<List<object>>

{

new List<object>{NonTerminal.OperatorsList}

});

Rules.Add(NonTerminal.VariableDeclaration, new List<List<object>>

{

new List<object>{Terminal.Var, NonTerminal.VariablesList}

});

Rules.Add(NonTerminal.VariablesList, new List<List<object>>

{

new List<object>{Terminal.Ident, NonTerminal.VariablesContinuation}

});

Rules.Add(NonTerminal.VariablesContinuation, new List<List<object>>

{

new List<object>{Terminal.Comma, NonTerminal.VariablesList},

new List<object>{}

});

Rules.Add(NonTerminal.OperatorsList, new List<List<object>>

{

new List<object>{NonTerminal.Operator, NonTerminal.OperatorsContinuation}

});

Rules.Add(NonTerminal.OperatorsContinuation, new List<List<object>>

{

new List<object>{NonTerminal.OperatorsList},

new List<object>{}

});

Rules.Add(NonTerminal.Operator, new List<List<object>>

{

new List<object>{NonTerminal.Assignment},

new List<object>{NonTerminal.ComplexOperator}

});

Rules.Add(NonTerminal.Assignment, new List<List<object>>

{

new List<object>{Terminal.Ident, Terminal.EqualSign, NonTerminal.Expression}

});

Rules.Add(NonTerminal.Expression, new List<List<object>>

{

new List<object>{Terminal.UnaryOperator, NonTerminal.Subexpression},

new List<object>{NonTerminal.Subexpression}

});

Rules.Add(NonTerminal.Subexpression, new List<List<object>>

{

new List<object>{Terminal.LeftBracket, NonTerminal.Expression, Terminal.RightBracket},

new List<object>{NonTerminal.BinaryOperatorSubexpression},

new List<object>{NonTerminal.Operand}

});

Rules.Add(NonTerminal.BinaryOperatorSubexpression, new List<List<object>>

{

new List<object>{Terminal.BinaryOperator, NonTerminal.Subexpression, NonTerminal.Subexpression }

});

Rules.Add(NonTerminal.Operand, new List<List<object>>

{

new List<object>{Terminal.Ident},

new List<object>{Terminal.Const}

});

Rules.Add(NonTerminal.ComplexOperator, new List<List<object>>

{

new List<object>{NonTerminal.CycleOperator},

new List<object>{NonTerminal.CompoundOperator}

});

Rules.Add(NonTerminal.CycleOperator, new List<List<object>>

{

new List<object>{Terminal.While,NonTerminal.Expression, Terminal.Do, NonTerminal.Operator}

});

Rules.Add(NonTerminal.CompoundOperator, new List<List<object>>

{

new List<object>{Terminal.Begin, NonTerminal.OperatorsList, Terminal.End}

});

}

}

}

# Выводы

В ходе данного курса нам удалось разработать лексический и семантический анализаторы. Разработанная программа принимает на вход файл с псевдокодом и печатает в консоль абстрактное синтаксическое дерево.

Разработанные парсер и лексер являются универсальными. Единственная проблема, которую не решает наша связка лексер + парсер – наличие цикличности в грамматике.

Таким образом, чтобы данная связка работала корректно, необходимо изменить исходную грамматику, чтобы исключить данную цикличность.

Мы решили данную проблему следующим образом:

*Исходная грамматика*

<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>

<Подвыражение> :: = ( <Выражение> ) | <Операнд> | <Подвыражение > <Бин.оп.> <Подвыражение>

*Измененная грамматика*

<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>

<Подвыражение> :: = ( <Выражение> ) | <Операнд> |<Бин.оп.> <Подвыражение > <Подвыражение>

Мы преобразовали все подвыражения из инфиксной формы в префиксную. Класс, отвечающий за преобразование, находится в приложении (ifmo.compilers.PrefixMaker).