МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Ульяновский государственный технический университет**»**

Кафедра «Вычислительная техника**»**

**Лабораторная работа №2**

по дисциплине «Системное программное обеспеченье»

«Синтаксический анализ»

Выполнил студент

гр. ИВТАСбд-31

Сутулов К. О.

Ульяновск, 2022

# Задание

Используя платформу .NET, необходимо реализовать синтаксический анализатор для языка программирования “C”. Синтаксический анализатор должен обрабатывать массив токенов, полученных в резульате работы лексического анализатора, реализованного в первой лабораторной работе.

# Ход работы

Синтаксический анализатор занимается построением абстрактного синтаксического дерева, в котором каждый узел согласован с определенной грамматикой языка программирования. При реализации синтаксического анализатора был выбран объектно-ориентированный метод нисходящего анализа.

Далее идет описание процесса реализации синтаксического анализатора.

Был создан класс SyntaxAnalyzer, отвечающий за генерацию синтаксического дерева по входящему в него потоку токенов, который передается ему через конструктор:

public class SyntaxAnalyzer

{

private TokenizedProgram \_tokenizedProgram { get; set; }

private Lexeme? \_currToken;

private Lexeme? \_nextToken;

private Lexeme? \_prevToken;

private int \_currTokenIndex = -1;

IdentifiersTable? top = null;

public SyntaxAnalyzer(TokenizedProgram tokenizedProgram)

{

\_tokenizedProgram = tokenizedProgram;

}

}

Затем был реализован метод NextToken, который позволяет получить следующий обрабатываемый токен, а также следующий и предыдущий токен относительно него:

private void NextToken()

{

if (\_currTokenIndex + 1 < \_tokenizedProgram.Lexemes.Count)

{

\_currTokenIndex++;

\_currToken = \_tokenizedProgram.Lexemes[\_currTokenIndex];

if (\_currTokenIndex + 1 < \_tokenizedProgram.Lexemes.Count)

{

\_nextToken = \_tokenizedProgram.Lexemes[\_currTokenIndex + 1];

}

else

{

\_nextToken = null;

}

if (\_currTokenIndex - 1 >= 0)

{

\_prevToken = \_tokenizedProgram.Lexemes[\_currTokenIndex - 1];

}

}

else

{

top = new IdentifiersTable(null);

Reset();

}

}

Метод CheckToken позволяет проверить токен на соответствие заданной строке:

private void NextToken()

{

if (\_currTokenIndex + 1 < \_tokenizedProgram.Lexemes.Count)

{

\_currTokenIndex++;

\_currToken = \_tokenizedProgram.Lexemes[\_currTokenIndex];

if (\_currTokenIndex + 1 < \_tokenizedProgram.Lexemes.Count)

{

\_nextToken = \_tokenizedProgram.Lexemes[\_currTokenIndex + 1];

}

else

{

\_nextToken = null;

}

if (\_currTokenIndex - 1 >= 0)

{

\_prevToken = \_tokenizedProgram.Lexemes[\_currTokenIndex - 1];

}

}

else

{

top = new IdentifiersTable(null);

Reset();

}

}

Далее был реализован метод GetStatement, который разбирает все доступные грамматики (по варианту лабораторной работы №1). Для каждой из грамматик был реализован собственный класс узла дерева. В данном методе также реализована проверка объявление переменных с помощью таблицы идентификаторов.

private StatementNode GetStatement()

{

if (\_currToken.Value == "{")

{

throw new Exception("invalid block");

}

if (\_currToken.Value == ";")

{

NextToken();

return null; //new NullNode();

}

if (\_currToken.Type == Lexeme.LexemeType.DataType)

{

string type = \_currToken.Value;

NextToken();

string name = \_currToken.Value;

if (top.Get(\_currToken) != null)

{

throw new Exception($"Variable {name} already declared");

}

if (\_currToken?.Type != Lexeme.LexemeType.Identifier)

{

throw new Exception($"Expected Identificator");

}

top.Put(\_currToken, new IdentifierNode(name, type));

DeclarationNode node = new DeclarationNode(name, type);

NextToken();

return node;

}

if (\_currToken.Value == "if")

{

NextToken();

CheckToken("(");

ExpressionNode condition = BoolOr();

CheckToken(")");

StatementNode then = GetBlock();

StatementNode? elseStatement = null;

if (\_currToken.Value == "else")

{

NextToken();

elseStatement = null;

if (\_currToken.Value == "{")

{

elseStatement = GetBlock();

} else if (\_currToken.Value == "if")

{

elseStatement = GetStatementSequense();

}

}

return new IfNode(condition, then, elseStatement);

}

if (\_currToken.Value == "do")

{

NextToken();

StatementNode body = GetBlock();

CheckToken("while");

CheckToken("(");

ExpressionNode cond = BoolOr();

CheckToken(")");

CheckToken(";");

return new DoWhileNode(body, cond);

}

if (\_currToken.Value == "++" || \_currToken.Value == "--" && \_prevToken.Type == Lexeme.LexemeType.Identifier)

{

string op = \_currToken.Value;

string id = \_prevToken.Value;

if (top.Get(\_prevToken) == null)

{

throw new Exception($"Variable {\_prevToken.Value} undefined");

}

NextToken();

CheckToken(";");

return new IncrementNode(op, id);

}

if (\_currToken.Value == "=")

{

return Assign();

}

if (\_currToken.Type == Lexeme.LexemeType.Identifier)

{

NextToken();

return GetStatement();

}

return null; //new NullNode();

}

Арифметические и логические выражения разбираются рекурсивно в порядке приоритетов операций. Разбор идет с операции “ИЛИ”, имеющей наименьший приоритет. Затем по порядку идут операции “И”, сравнение, сложение и вычитание, деление и умножение, отрицание и унарный минус. Наивысшим приоритетом обладают константы и значения идентификаторов.

private ExpressionNode BoolOr()

{

ExpressionNode expression = BoolAnd();

while (\_currToken.Value == "||")

{

string op = \_currToken.Value;

NextToken();

expression = new ArithmeticNode(op, expression, BoolAnd());

}

return expression;

}

private ExpressionNode BoolAnd()

{

ExpressionNode expression = BoolMatch();

while (\_currToken.Value == "&&")

{

string op = \_currToken.Value;

NextToken();

expression = new ArithmeticNode(op, expression, BoolMatch());

}

return expression;

}

private ExpressionNode BoolMatch()

{

ExpressionNode expression = Expression();

while (\_currToken.Value == "==" || \_currToken.Value == ">" || \_currToken.Value == "<" || \_currToken.Value == "<=" || \_currToken.Value == ">=" || \_currToken.Value == "!=")

{

string op = \_currToken.Value;

NextToken();

expression = new ArithmeticNode(op, expression, Expression());

}

return expression;

}

private ExpressionNode Expression()

{

ExpressionNode expression = Term();

while (\_currToken.Value == "+" ||

\_currToken.Value == "-")

{

string op = \_currToken.Value;

NextToken();

expression = new ArithmeticNode(op, expression, Term());

}

return expression;

}

private ExpressionNode Term()

{

ExpressionNode expression = Unary();

while (\_currToken.Value == "\*" ||

\_currToken.Value == "/")

{

string op = \_currToken.Value;

NextToken();

expression = new ArithmeticNode(op, expression, Unary());

}

return expression;

}

private ExpressionNode Unary()

{

if (\_currToken.Value == "-")

{

NextToken();

return new UnaryNode("-", Unary());

}

else if (\_currToken.Value == "!")

{

string op = \_currToken.Value;

NextToken();

return new NotNode(op, Unary());

}

else

{

return Factor();

}

}

private ExpressionNode Factor()

{

ExpressionNode expressionNode = null;

if (\_currToken.Type == Lexeme.LexemeType.Constant)

{

expressionNode = new ConstantNode(\_currToken.Value, \_currToken.Attributes["Type"]);

NextToken();

return expressionNode;

}

if (\_currToken.Type == Lexeme.LexemeType.Identifier)

{

IdentifierNode identifier = top.Get(\_currToken);

if (identifier == null)

{

throw new Exception($"undefined variable {\_currToken.Value}");

}

NextToken();

return identifier;

}

if (\_currToken.Value == "(")

{

NextToken();

expressionNode = BoolOr();

CheckToken(")");

return expressionNode;

}

throw new Exception("Syntax error");

}

}

Для обработки блока операций реализованы методы GetBlock и GetStatementSequence. В методе GetBlock также происходит создание новой таблицы идентификаторов с сохранением ссылки на предыдущую. Метод GetStatementSequence рекурсивно вызывает метод GetStatement, рекурсия остановится, когда метод достигнет конца блока:

private StatementNode GetStatementSequense()

{

if (\_currToken == null || \_currToken.Value == "}") return null; //new NullNode();

while (\_currToken.Value == ";") NextToken();

return new SequenceNode(GetStatement(), GetStatementSequense());

}

private StatementNode GetBlock()

{

CheckToken("{");

IdentifiersTable table = top;

top = new IdentifiersTable(top);

StatementNode node = GetStatementSequense();

CheckToken("}");

top = table;

return node;

}

Для обработки операции присвоения используется метод Assign. В данном методе также существует проверка на необъявленный идентификатор:

private StatementNode Assign()

{

if (\_prevToken.Type != Lexeme.LexemeType.Identifier) throw new Exception("Syntax error");

string name = \_prevToken.Value;

IdentifierNode identifier = top.Get(\_prevToken);

if (identifier == null)

{

throw new Exception($"undefined variable {name}");

}

NextToken();

SetNode statement = new SetNode(name, BoolOr());

CheckToken(";");

return statement;

}

Наконец был реализован публичный метод GenerateTree, который позволяет возвращает корневой элемент дерева:

public StatementNode GenerateTree()

{

Reset();

NextToken();

StatementNode node = GetBlock();

return node;

}

Синтаксический анализатор реализован.

# Тестирование программы

Для тестирования программы был написан пример на языке C, который охватывает все доступные синтаксическому анализатору грамматики:

{

int a = (100 + 2) \* 5;

int b;

b = -5;

do {

int c = a + 2;

if (a + 2 == c) {

c = 5;

} else {

do {

c++;

} while(c + 2 \* (5 - 8) + 4 < (3 \* 100-5) \* 2);

}

} while(true);

}

Для данного примера было успешно сгенерировано синтаксическое дерево, структура которого соответствует исходной программе. Вывод синтаксического дерева производится в формате JSON и представлен в приложении.

Обработка ошибок также выполняется корректно. Например, для следующего примера

{

int a = (100 + 2) \* 5;

b = -5;

}

было выброшено исключение о необъявленной переменной:

**System.Exception:** "undefined variable b"

# Вывод

В процессе выполнения лабораторной работы с использованием платформы .NET была составлена программная реализация синтаксического анализатора языка “С” с использованием нисходящего метода синтаксического анализа.

# Приложение: вывод синтаксического дерева

{

"?xml": {

"@version": "1.0",

"@encoding": "utf-16"

},

"SequenceNode": {

"@xmlns:xsi": "http://www.w3.org/2001/XMLSchema-instance",

"@xmlns:xsd": "http://www.w3.org/2001/XMLSchema",

"Statement": {

"@xsi:type": "DeclarationNode",

"Identificator": "a",

"Type": "int"

},

"Next": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "SetNode",

"Name": "a",

"Expression": {

"@xsi:type": "ArithmeticNode",

"Operation": "\*",

"Expression1": {

"@xsi:type": "ArithmeticNode",

"Operation": "+",

"Expression1": {

"@xsi:type": "ConstantNode",

"Operation": "100",

"type": "int"

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "2",

"type": "int"

}

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "5",

"type": "int"

}

}

},

"Next": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "DeclarationNode",

"Identificator": "b",

"Type": "int"

},

"Next": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "SetNode",

"Name": "b",

"Expression": {

"@xsi:type": "UnaryNode",

"Operation": "-",

"Expression": {

"@xsi:type": "ConstantNode",

"Operation": "5",

"type": "int"

}

}

},

"Next": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "DoWhileNode",

"body": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "DeclarationNode",

"Identificator": "c",

"Type": "int"

},

"Next": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "SetNode",

"Name": "c",

"Expression": {

"@xsi:type": "ArithmeticNode",

"Operation": "+",

"Expression1": {

"@xsi:type": "IdentifierNode",

"Operation": "a",

"Type": "int"

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "2",

"type": "int"

}

}

},

"Next": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "IfNode",

"Condition": {

"@xsi:type": "ArithmeticNode",

"Operation": "==",

"Expression1": {

"@xsi:type": "ArithmeticNode",

"Operation": "+",

"Expression1": {

"@xsi:type": "IdentifierNode",

"Operation": "a",

"Type": "int"

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "2",

"type": "int"

}

},

"Expression2": {

"@xsi:type": "IdentifierNode",

"Operation": "c",

"Type": "int"

}

},

"Then": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "SetNode",

"Name": "c",

"Expression": {

"@xsi:type": "ConstantNode",

"Operation": "5",

"type": "int"

}

}

},

"Else": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "DoWhileNode",

"body": {

"@xsi:type": "SequenceNode",

"Statement": {

"@xsi:type": "IncrementNode",

"Operation": "++",

"Identifier": "c"

}

},

"condition": {

"@xsi:type": "ArithmeticNode",

"Operation": "<",

"Expression1": {

"@xsi:type": "ArithmeticNode",

"Operation": "+",

"Expression1": {

"@xsi:type": "ArithmeticNode",

"Operation": "+",

"Expression1": {

"@xsi:type": "IdentifierNode",

"Operation": "c",

"Type": "int"

},

"Expression2": {

"@xsi:type": "ArithmeticNode",

"Operation": "\*",

"Expression1": {

"@xsi:type": "ConstantNode",

"Operation": "2",

"type": "int"

},

"Expression2": {

"@xsi:type": "ArithmeticNode",

"Operation": "-",

"Expression1": {

"@xsi:type": "ConstantNode",

"Operation": "5",

"type": "int"

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "8",

"type": "int"

}

}

}

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "4",

"type": "int"

}

},

"Expression2": {

"@xsi:type": "ArithmeticNode",

"Operation": "\*",

"Expression1": {

"@xsi:type": "ArithmeticNode",

"Operation": "-",

"Expression1": {

"@xsi:type": "ArithmeticNode",

"Operation": "\*",

"Expression1": {

"@xsi:type": "ConstantNode",

"Operation": "3",

"type": "int"

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "100",

"type": "int"

}

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "5",

"type": "int"

}

},

"Expression2": {

"@xsi:type": "ConstantNode",

"Operation": "2",

"type": "int"

}

}

}

}

}

}

}

}

},

"condition": {

"@xsi:type": "ConstantNode",

"Operation": "true",

"type": "bool"

}

}

}

}

}

}

}

}