Учреждение образования

“БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ”

Кафедра информатики

Отчет по лабораторной работе №3

Синтаксический анализатор.

Выполнил:
Корзун А. Д.

Проверил:
Шиманский В. В.

Минск 2022

1. Постановка задачи:

В данной работе ставится задача исследования области синтаксического анализатора

Основной целью работы является построение дерева разбора (синтаксического дерева), которое отражает синтаксическую структуру входной последовательности. Данная лабораторная работа является продолжением лексического анализатора и дополнена классом Parser. Таким образом на основе анализа выражений, состоящих из литералов, операторов и круглых скобок выполняется группирование токенов исходной программы в грамматические фразы, используемые для синтеза вывода.

2. Теоретические сведения:

В ходе синтаксического анализа исходный текст программы проверяется на на соответствие синтаксическим нормам языка с построением дерево разбора (синтаксическое дерево), которое отражает синтаксическую структуру входной последовательности и удобно для дальнейшего использования, а также в случае несоответствия – позволяет вывести сообщения об ошибках.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

Таким образом на основе анализа выражений, состоящих из литералов, операторов и круглых скобок выполняется группирование токенов исходной программы в грамматические фразы, используемые для синтеза вывода.

3. Результат работы:

Анализируемый код:

```
_list = list(1, 2, 3, 4)
b = 5
if b == 10:
        print('10')
elif b == 11:
        print('11')
else:
        print(b)
_list( 3 ) = 5
s = 0
while i < 4:
         s =  s+ _lsit( i )
         i= i  + 1
print('sum: ')
print(s)
_list_2 = list(0)
while i < 100:
        _lsit_2( i ) = i
        i = i + 1
print(_list_2)
val = 5
def inc_val(val) :
        val = val + 10
        print(vaaaal)
inc_val(val)
```

Полученное дерево:

/ =
/\ _list
/\ list
/\\ ,
/\\\ ,
/\\\\ 1
/\\\\ 2
/\\\ ,
/\\\\ 3
/\\\\ 4
/
/ =

/\ b
/\ 5
/
/ if
/\ ==
/\\ b
/\\ 10
/
//| print
//|\ '10'
//|
/ elif
/\ ==
/\\ b
/\\ 11
/
//| print
//|\ '11'
//|
/ else
/\ :
/
//| print
//|\ b
//|
/ _list
/\ =
/\\ 3
/\\ 5
/
/ =
/\ s
/\ 0
/
/ while
/\ <
/\\ i
/\\ 4
/
//| =
//|\ s
//|\ +

```
/ / \ \ s
/ / \ \ _lsit
/ / \ \ \ i
/ /
/ / =
/ / \ i
/ / \ +
/ / \ \ i
/ / \ \ 1
/ /
| print
/ \ 'sum: '
/
| print
/ \ s
/
| =
/ \ _list_2
/ \ list
/ \ \ 0
/
| while
/ \ <
/ \ \ i
/ \ \ 100
/
/ / _lsit_2
/ / \ =
/ / \ \ i
/ / \ \ i
/ /
/ / =
/ / \ i
/ / \ +
/ / \ \ i
/ / \ \ 1
/ /
| print
/ \ _list_2
/
| =
/ \ val
```

```
/\ 5
|
| def
/\ inc_val
/\\ :
/\\\ val
|
|| =
||\ val
||\ +
||\\ val
||\\ 10
||
|| print
||\ vaaaal
||
| inc_val
/\ val
|
```

4. Разбор ошибок

print(s)()(

```
line 15 char 8 :: UNEXPECTED_TOKEN
print(s)()(
       ^
```

_list_2 = list(0 4 10)

```
SYNTAX ERROR unknown operator
line 15 char 18:
 list_2 = list(0 4 10)
```

print(s) 12312312

```
line 14 char 8:
print(s) 12312312
       ^
```

print(s)))

```
line 15 char 8 :: UNEXPECTED_TOKEN
print(s)))
       ^
```

_list_2 = list(,0)

```
line 17 char 15 :: UNEXPECTED_TOKEN
_list_2 = list(,0)
               ^
```

5. Выводы

В результате выполнения лабораторной работы была написана программа, выполняющая алгоритм построения синтаксического дерева, отлавливающая синтаксические ошибки.

Приложение. Код программы

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;
using static ConsoleApp1.Token;
using static ConsoleApp1.LexicalAnalizer;
using ConsoleTables;
namespace ConsoleApp1
{
    class Program
    {
        static void PrintTokensDictionary(Dictionary<string, Token> dictionary)
        {
            ConsoleTable consoleTable = new ConsoleTable("TOKEN",
"DESCRIPTION");
            foreach (Token token in dictionary.Values)
            {
                consoleTable.AddRow(token.Value, token.DescriptionString);
            }
            consoleTable.Write();
        }
        static string PrintNodeWithChildren(SyntaxAnalizer.ExpressionNode node,
string indentation)
        {
            if (node == null)
            {
                return "";
            }
            SyntaxAnalizer.ValidateNode(node);
            StringBuilder stringBuilder = new StringBuilder();
            stringBuilder.AppendLine($"{indentation} {node.Operator.Value}");
            if (node.Left != null)
                stringBuilder.Append(PrintNodeWithChildren(node.Left, indentation +
"\\"));
```

```
            if (node.Right != null)
                stringBuilder.Append(PrintNodeWithChildren(node.Right, indentation +
"\\"));
            return stringBuilder.ToString();
        }

        static void PrintSyntaxTree(IEnumerable<SyntaxAnalizer.ExpressionNode>
nodes, int nestingLevel = 1)
        {
            string indentation = new String('|', nestingLevel);
            foreach (var node in nodes)
            {

                Console.Write(PrintNodeWithChildren(node, indentation));
                Console.WriteLine(indentation);
                PrintSyntaxTree(node.Block, nestingLevel+1);
            }
        }
        static string errorDescription(int indexInCodeLine, string codeLine)
        {
            StringBuilder stringBuilder = new StringBuilder(codeLine);
            stringBuilder.AppendLine();
            stringBuilder.Append(new string(' ', indexInCodeLine));
            stringBuilder.Append('^');
            return stringBuilder.ToString();
        }
        static void DoTheJob(IEnumerable<string> codeLines)
        {
            Dictionary<string, Token> constants = new Dictionary<string, Token>();
            Dictionary<string, Token> variables = new Dictionary<string, Token>();
            Dictionary<string, Token> operators = new Dictionary<string, Token>();
            Dictionary<string, Token> keywords = new Dictionary<string, Token>();
            List<LexicalError> errors = new List<LexicalError>();
            TreeList<SyntaxAnalizer.ExpressionNode> tree = new
TreeList<SyntaxAnalizer.ExpressionNode>(null);
            TreeList<SyntaxAnalizer.ExpressionNode> currentBlock = tree;
            int lineNumber = 0;
            SyntaxAnalizer sa = new SyntaxAnalizer();
            LexicalAnalizer la = new LexicalAnalizer();
```

```
            int previousLineIndentation = 0;
            foreach (string line in codeLines)
            {
                Construction construction = la.AnaliseLine(line, lineNumber);
                if (construction.Tokens.Count == 0)
                {
                    lineNumber++;
                    continue;
                }
                for (int i = 0; i < construction.Tokens.Count; i++)
                {
                    Token token = construction.Tokens[i];
                    if (token.IsReservedIdToken)
                        keywords.TryAdd(token.Value, token);
                    else if (token.IsOperation)
                        operators.TryAdd(token.Value, token);
                    else if (token.IsConstant)
                        constants.TryAdd(token.Value, token);
                    else if (token.TokenType != TokenTypes.UNKNOWN)
                    {
                        variables.TryAdd(token.Value, token);
                    }
                }
                if (construction.HasErrors)
                {
                    Console.ForegroundColor = ConsoleColor.Red;
                    Console.WriteLine("\t\t ERRORS");
                    Console.ResetColor();
                    foreach (LexicalError error in construction.Errors)
                    {
                        Console.WriteLine($"line {error.CodeLineNumber + 1} char
{error.IndexInCodeLine + 1} :: {error.ErrorType}");
                        Console.WriteLine(error.Description);
                    }
                    Console.Read();
                    Environment.Exit(1);
                }
                SyntaxAnalizer.ExpressionNode node = null;
                bool isElifElseNode = false;
                bool newBlockToOpen = false;
```

```
            node = sa.Analyse(construction.Tokens, out newBlockToOpen, out
isElifElseNode);
            int indentationDiff = previousLineIndentation - construction.Indentation;
                    if (indentationDiff > 0)
            {
               for (int i = previousLineIndentation-1; i >= construction.Indentation;
i--)
               {
                  currentBlock = currentBlock.Parent;
                  if (currentBlock.Indentation == i)
                     break;
               }
               // currentBlock = currentBlock.Parent; // TODO: create parent
relationship between BLOCKS to support >1 level nesting
               if (node.Operator.IsElif && !currentBlock.Last().Operator.IsIf)
               {
                  throw new SyntaxAnalizer.SyntaxErrorException(
                        "elif block not allowed here",
                        node.Operator.Value,
                        node.Operator.CodeLineIndex,
                        node.Operator.CodeLineNumber
                     );
               }
               else if (node.Operator.IsElse && !(currentBlock.Last().Operator.IsIf ||
currentBlock.Last().Operator.IsElif))
               {
                  throw new SyntaxAnalizer.SyntaxErrorException(
                        "else block not allowed here",
                        line,
                        node.Operator.CodeLineIndex,
                        node.Operator.CodeLineNumber
                     );
               }
            }
            previousLineIndentation = construction.Indentation;
            lineNumber++;
            if (newBlockToOpen)
            {
```

```
                if ((node.Operator.IsElif || node.Operator.IsElse) &&
!currentBlock.Last().Operator.IsIf && !currentBlock.Last().Operator.IsElif)
                {
                    throw new SyntaxAnalizer.SyntaxErrorException(
                        "lacks IF clause for elif|else block to appear",
                        node.Operator.Value,
                        node.Operator.CodeLineIndex,
                        node.Operator.CodeLineNumber
                        );
                }
                currentBlock.Add(node);
                currentBlock.Last().Block = new
TreeList<SyntaxAnalizer.ExpressionNode>(currentBlock);
                currentBlock = currentBlock.Last().Block;
                currentBlock.Indentation = construction.Indentation;
                continue;
            }
            currentBlock.Add(node);
        }
        if (errors.Any())
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine("\t\t ERRORS");
            Console.ResetColor();
            foreach (LexicalError error in errors)
            {
                Console.WriteLine($"line {error.CodeLineNumber + 1} char
{error.IndexInCodeLine + 1} :: {error.ErrorType}");
                Console.WriteLine(error.Description);
            }
        }
        Console.WriteLine("SYNTAX TREE:\n");
        PrintSyntaxTree(tree);
        // console tables output block
        Console.WriteLine("\n \t\t CONSTANTS");
        PrintTokensDictionary(constants);

        Console.WriteLine("\n \t\t VARIABLES");
        PrintTokensDictionary(variables);
```

```
            Console.WriteLine("\n \t\t KEYWORS");
            PrintTokensDictionary(keywords);
            Console.WriteLine("\n \t\t OPERATORS");
            PrintTokensDictionary(operators);
        }
        static void Main(string[] args)
        {
            Console.OutputEncoding = System.Text.Encoding.UTF8;
            string FILENAME = @"test.py";
            IEnumerable<string> codeLines =
System.IO.File.ReadLines(FILENAME);
            try
            {
                DoTheJob(codeLines);
            }
            catch (SyntaxAnalizer.SyntaxErrorException e)
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine($"SYNTAX ERROR {e.Message}");
                Console.ResetColor();
                Console.WriteLine($"line {e.LineNumber} char {e.PositionInLine}:");
                Console.WriteLine(errorDescription(e.PositionInLine,
codeLines.ElementAt(e.LineNumber).Trim()));
            }
            catch (InvalidOperationException e)
            {
                Console.WriteLine($"SYNTAX ERROR {e.Message}");
                Console.WriteLine("block opening element has nothing in its block!");
            }
            Console.Read();
        }
    }
}


namespace ConsoleApp1
{
    class SyntaxAnalizer
    {
        protected int OpenedBracketsLevel = 0;
```

```
        protected int CurrentBlockLevel = 0;

        public ExpressionNode Analyse(IEnumerable<Token> tokens, out bool
startNewBlock, out bool isElifElseNode)
        {
            OpenedBracketsLevel = 0;
            startNewBlock = false;
            isElifElseNode = false;
            var firstToken = tokens.FirstOrDefault();
                    if (firstToken?.IsBlockOpeningOperation == true)
            {
                startNewBlock = true;
                isElifElseNode = firstToken.TokenType == TokenTypes.ELSE ||
firstToken.TokenType == TokenTypes.ELIF;
                if (tokens.LastOrDefault()?.TokenType != Token.TokenTypes.COLON)
                {
                    var t = tokens.LastOrDefault();
                    throw new SyntaxErrorException("colon expected", t.Value,
t.CodeLineIndex, t.CodeLineNumber);
                }
            }
            ExpressionNode root = BuildTree(tokens);
            if (OpenedBracketsLevel != 0)
            {
                throw new SyntaxErrorException("brackets do not match",
tokens.Last().Value, tokens.Last().CodeLineIndex,
tokens.Last().CodeLineNumber);
            }
            return root;
        }
        protected ExpressionNode BuildTree(IEnumerable<Token> tokens,
ExpressionNode parent = null)
        {
            ExpressionNode root = null;
            ExpressionNode left = null;

            Token token = tokens.FirstOrDefault();
            if (token is null)
                return null;
```

```
            if (token.IsConstant || token.TokenType == Token.TokenTypes.ID ||
token.TokenType == Token.TokenTypes.BUILT_IN_FUNCTION)
            {
                left = new ExpressionNode()
                {
                    Operator = token,
                    Type =
ExpressionNode.TokensToExpressionTypes.GetOrDefault(token.TokenType,
ExpressionNode.ExpressionTypes.UNKNOWN)
                };
                var tt = tokens.ElementAtOrDefault(1)?.TokenType;
                if (tt == Token.TokenTypes.OPENING_ROUND_BRACKET)
                {
                    root = left;
                    left = null;
                    root.Type = ExpressionNode.ExpressionTypes.FUNCTION_CALL;
                    root.Right = BuildTree(tokens.Skip(1));
                }
                else if (tt == Token.TokenTypes.COLON)
                {
                    root = left;
                    left = null;
                    root.Right = BuildTree(tokens.Skip(2));
                                }
                else
                {
                    root = BuildTree(tokens.Skip(1));
                    left.Parent = root;
                }
            }
            else if (token.IsOpeningBracket)
            {
                this.OpenedBracketsLevel++;
                root = BuildTree(tokens.Skip(1));
                root.OperatorPriority++;
            }
            else if (token.IsClosingBracket)
            {
                this.OpenedBracketsLevel--;
                root = BuildTree(tokens.Skip(1));
```

```
            if (root != null)
                root.OperatorPriority--;
            }
            else if (token.IsOperation)
            {
                root = new ExpressionNode()
                {
                    Operator = token,
                    Type =
ExpressionNode.TokensToExpressionTypes.GetOrDefault(token.TokenType,
ExpressionNode.ExpressionTypes.UNKNOWN)
                };
                if (token.TokenType == Token.TokenTypes.MULTIPLICATION ||
token.TokenType == Token.TokenTypes.DIVISION)
                {
                    root.OperatorPriority++;
                }
                root.Right = BuildTree(tokens.Skip(1), root);
            }
            if (root is null)
            {
                if (left is null)
                    return null;
                left.Parent = parent;
                return left;
            }
            root.Parent = parent;
            if (left != null)
                root.InsertDeepLeft(left);

            if (root.Right != null && root.Operator.IsOperation &&
root.Right.Operator.IsOperation && root.OperatorPriority >
root.Right.OperatorPriority)
                return root.LeftRotation();
            return root;
        }
        public static ExpressionNode ValidateNode(ExpressionNode node)
        {
            switch (node.Type)
            {
```

```
            case ExpressionNode.ExpressionTypes.BINARY_OPERATION:
                                if (node.Left == null || node.Right == null)
                {
                   throw new SyntaxErrorException(
                      "binary operation lacks operand",
                      node.Operator.Value,
                      node.Operator.CodeLineIndex,
                      node.Operator.CodeLineNumber
                      );
                }
                break;
                        case
ExpressionNode.ExpressionTypes.BLOCK_OPENING_CONDITIONAL_OPERA
TION:
                                if (node.Left != null || node.Right == null)
                   throw new SyntaxErrorException(
                      "conditional operator wrong usage",
                      node.Operator.Value,
                      node.Operator.CodeLineIndex,
                      node.Operator.CodeLineNumber
                      );
                break;
                        case ExpressionNode.ExpressionTypes.UNKNOWN:
               throw new SyntaxErrorException(
                  "unknown expression",
                  node.Operator.Value,
                  node.Operator.CodeLineIndex,
                  node.Operator.CodeLineNumber
                  );
            case ExpressionNode.ExpressionTypes.OPERAND:
               if (node.Left != null)
                  throw new SyntaxErrorException(
                  "unknown operator",
                  node.Operator.Value,
                  node.Operator.CodeLineIndex,
                  node.Operator.CodeLineNumber
                  );
               break;
            default:
               break;
```

```
            }
            return node;
        }
        public class SyntaxErrorException : FormatException
        {
            public string Value { get; set; }
            public int PositionInLine { get; set; }
            public int LineNumber { get; set; }
            public SyntaxErrorException(string message, string value, int
positionInLine, int lineNumber) : base(message)
            {
                Value = value;
                PositionInLine = positionInLine;
                LineNumber = lineNumber;
            }
        }

        public class ExpressionNode
        {
            public ExpressionNode Left = null;
            public Token Operator = null;
            public ExpressionTypes Type;
            public int OperatorPriority = 0;
            public ExpressionNode Right = null;
            public ExpressionNode Parent = null;
            public TreeList<ExpressionNode> Block = new
TreeList<ExpressionNode>(null);
            public ExpressionNode LeftRotation()
            {
                ExpressionNode newRoot = new ExpressionNode()
                {
                    Right = this.Right.Right,
                    Operator = this.Right.Operator,
                    Type = this.Right.Type,
                    Parent = this.Parent
                };
                newRoot.Left = new ExpressionNode()
                {
                    Left = this.Left,
                    Right = this.Right.Left,
```

```
                Operator = this.Operator,
                Type = this.Type,
                Parent = newRoot
            };
            return newRoot;
        }
        public void InsertDeepLeft(ExpressionNode node)
        {
            ExpressionNode temp = this;
            while (!(temp.Left is null))
            {
                temp = temp.Left;
            }
            temp.Left = node;
        }
        public override string ToString()
        {
            return $"({Operator.ToString()})";
        }
        public enum ExpressionTypes
        {
            UNKNOWN,
            UNARY_OPERATION,
            BINARY_OPERATION,
            BLOCK_OPENING_CONDITIONAL_OPERATION,
            BLOCK_OPENING_OPERATION,
            FUNCTION_CALL,
            FUNCTION_DEF,
            OPERAND
        };
        public static Dictionary<TokenTypes, ExpressionTypes>
TokensToExpressionTypes = new Dictionary<TokenTypes, ExpressionTypes>()
        {
            [TokenTypes.ASSIGN] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.COMMA] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.DOT] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.IF] =
ExpressionTypes.BLOCK_OPENING_CONDITIONAL_OPERATION,
            [TokenTypes.ELIF] =
ExpressionTypes.BLOCK_OPENING_CONDITIONAL_OPERATION,
```

```
            [TokenTypes.ELSE] =
ExpressionTypes.BLOCK_OPENING_OPERATION,
            [TokenTypes.FOR] =
ExpressionTypes.BLOCK_OPENING_CONDITIONAL_OPERATION,
            [TokenTypes.WHILE] =
ExpressionTypes.BLOCK_OPENING_CONDITIONAL_OPERATION,
            [TokenTypes.PLUS] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.MINUS] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.MODULE] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.DIVISION] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.MULTIPLICATION] =
ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.NOT] = ExpressionTypes.UNARY_OPERATION,
            [TokenTypes.AND] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.OR] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.IN] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.LOWER] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.LOWER_OR_EQUAL] =
ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.GREATER] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.GREATER_OR_EQUAL] =
ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.NOT_EQUAL] =
ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.EQUAL] = ExpressionTypes.BINARY_OPERATION,
            [TokenTypes.FUNCTION_DEFINITION] =
ExpressionTypes.FUNCTION_DEF,
            [TokenTypes.STRING_CONST] = ExpressionTypes.OPERAND,
            [TokenTypes.INT_NUM] = ExpressionTypes.OPERAND,
            [TokenTypes.FLOAT_NUM] = ExpressionTypes.OPERAND,
            [TokenTypes.ID] = ExpressionTypes.OPERAND,
            [TokenTypes.BUILT_IN_FUNCTION] = ExpressionTypes.OPERAND,
            [TokenTypes.COLON] =
ExpressionTypes.BLOCK_OPENING_OPERATION
        };
    }
}
```