Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

“Лексический анализатор.”

Выполнил:

студент гр. 953501

Кондрашов И.Д.

Проверил:

Ассистент кафедры информатики

Шиманский В. В.

Минск 2022

Содержание

1. **Постановка задачи**
2. **Теория**
3. **Созданный лексический анализатор**
4. **Пример работы программы**
5. **Вывод**

**Примечание. Код программы**

1. **Постановка задачи**

Освоение работы с существующими лексическими анализаторами (по желанию). Разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1.Определяются лексические правила. Выполняется перевод потока символов в поток лексем (токенов).

Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называемые лексемами.

На вход программы подается текстовый файл (напр. с именем INPUT.TXT), содержащий строки символов анализируемой программы. Например, строка может задавать переменной значения арифметического выражения в виде

*ПЕРЕМЕННАЯ* = *ВЫРАЖЕНИЕ*.

Выражение может включать:

* Знаки сложения и умножения («+» и «\*»);
* Круглые скобки («(» и «)»);
* Константы (например, 5; 3.8; 1e+18, 8.41E–10);
* Имена переменных.

Имя переменной – это последовательность букв и цифр, начинающаяся с буквы.

Разбор выражения *COST* = (*PRICE*+*TAX*)\*0.98.

Проанализируем выражение:

* *COST*, *PRICE* и *TAX* – лексемы-идентификаторы;
* 0.98 – лексема-константа; =, +, \* – просто лексемы.

Пусть все константы и идентификаторы можно отображать в лексемы типа <идентификатор> (<ИД>). Тогда выходом лексического анализатора будет последовательность лексем <ИД1>=(<ИД2>+<ИД3>)\*<ИД4>.

Вторая часть компоненты лексемы (указатель, т.е. номер лексемы в таблице имен) – показана в виде индексов. Символы «=», «+» и «\*» трактуются как лексемы, тип которых представляется ими самими. Они не имеют связанных с ними данных и, следовательно, не имеют указателей.

#### РАБОТА С ТАБЛИЦЕЙ ИМЕН

После того, как в результате лексического анализа лексемы распознаны, информация о некоторых из них собирается и записывается в таблицу имен.

Для нашего примера *COST*, *PRICE* и *TAX* – переменные с плавающей точкой. Рассмотрим вариант такой таблицы. В ней перечислены все идентификаторы вместе с относящейся к ним информацией (табл. 1).

Табл. 1 – Таблица имен

|  |  |  |
| --- | --- | --- |
| Номер элемента | Идентификатор | Информация |
| 1 | *COST* | Переменная с плавающей точкой |
| 2 | *PRICE* | Переменная с плавающей точкой |
| 3 | *TAX* | Переменная с плавающей точкой |
| 4 | 0.98 | Константа с плавающей точкой |

Если позднее во входной цепочке попадается идентификатор, надо справиться в этой таблице, не появлялся ли он ранее. Если да, то лексема, соответствующая новому вхождению этого идентификатора, будет той же, что и у предыдущего вхождения.

Возможно создание таблиц констант, ключевых слов, разделителей, логических/математических операторов

При определении неверной последовательности символов, необходимо обнаружить эту ошибку и выдать сообщение о ней. Всего необходимо показать скриншоты нахождения 4-х **лексических** ошибок.

1. **Теория**

Лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Обычно при работе лексический анализатор взаимодействует также с таблицей символов. Когда лексический анализатор встречается с лексемой, составляющей идентификатор, эту лексему требуется внести в таблицу символов. В некоторых случаях лексический анализатор может получать из таблицы символов некоторую информацию об идентификаторах, которая может помочь ему верно определить передаваемый синтаксическому анализатору токен.

Поскольку лексический анализатор является частью компилятора, которая читает исходный текст, он может заодно выполнять и некоторые другие действия, помимо идентификации лексем. Одной из таких задач является отбрасывание комментариев и пробельных символов (пробел, символы табуляции и новой строки, а также, возможно, некоторые другие символы, использующиеся для отделения токенов друг от друга во входном потоке). Еще одной задачей является синхронизация сообщений об ошибках, генерируемых компилятором, с исходной программой. Например, лексический анализатор может отслеживать количество символов новой строки, чтобы каждое сообщение об ошибке сопровождалось номером строки, в которой она обнаружена. В некоторых компиляторах лексический анализатор создает копию исходной программы с сообщениями об ошибках, вставленными в соответствующие места исходного текста.

1. **Созданный лексический анализатор**

Код исследуемой программы в файле подается на вход программы-анализатора и считывается построчно. С помощью строки регулярных выражений в каждой строке выделяются следующие типы токенов:

* Комментарии
* Строковые константы
* Целочисленные константы
* Константы типа float
* Символы-операторы, слова (потенциальные ID).

Таким образом ни один символ входного файла не будет потерян из-за несоответствия какой-то части заданных правил регулярных выражений.

1. **Примеры работы программы**

**Пример 1. Пирамидальная сортировка**

def heapify(nums, heap, root):  
 largest = root  
 left = (2 \* root) + 1  
 right = (2 \* root) + 2  
 if left < heap and nums[left] > nums[largest]:  
 largest = left  
 if right < heap and nums[right] > nums[largest]:  
 largest = right  
 if largest != root:  
 nums[root], nums[largest] = nums[largest], nums[root]  
 heapify(nums, heap, largest)  
  
def heap\_sort(nums):  
 n = len(nums)  
 for i in range(n, -1, -1):  
 heapify(nums, n, i)  
 for i in range(n - 1, 0, -1):  
 nums[i], nums[0] = nums[0], nums[i]  
 heapify(nums, i, 0)  
  
def test1():  
 random\_nums = [46, 18, 1, 35, 26]  
 heap\_sort(random\_nums)  
 print('Пирамидальная сортировка')  
 print(random\_nums)  
  
test1()

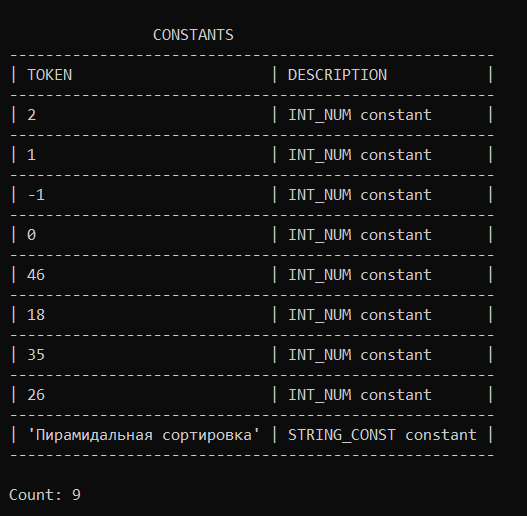


Рис 1. Первая часть вывода

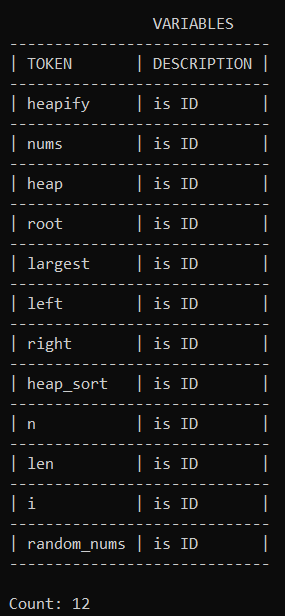


Рис 2. Вторая часть вывода

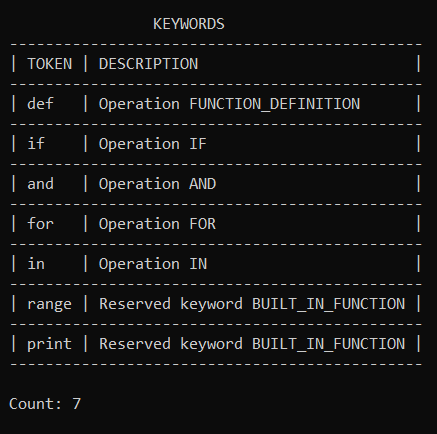


Рис 3. Третья часть вывода

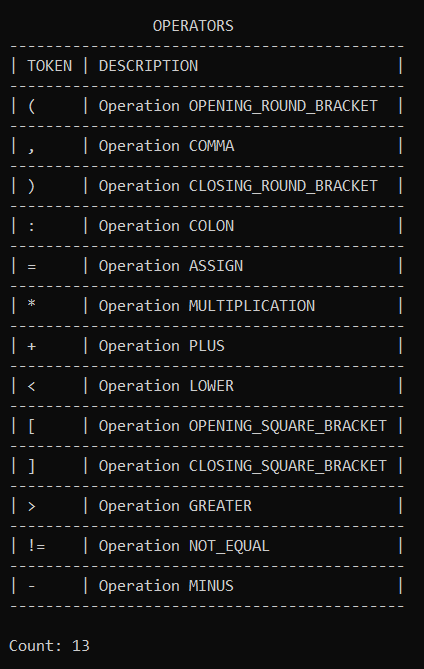


Рис 4. Четвертая часть вывода

**Пример 2. Пирамидальная сортировка с ошибками**

def heapify(nums, heap, root) :  
 largest === root ‘Ошибка №1’  
 left = (2 \* root) + 1

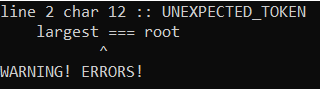


Рис 5. Вывод информации о найденной ошибке

left = (2 \* root) +- 1 ‘Ошибка №2’  
right = (2 \* root) + 2

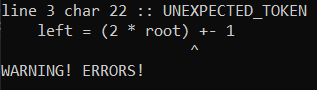


Рис 6. Вывод информации о найденной ошибке

if right < heap and nums[right] > nums[largest]; ‘Ошибка №3’  
 largest = right

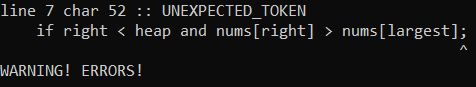


Рис 7. Вывод информации о найденной ошибке

def heap\_sort(nums) :  
 n = len(nums)  
 fo№r I in range(n, -1, -1) : ‘Ошибка №4’  
 heapify(nums, n, i)

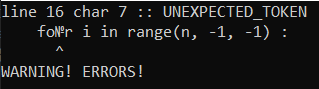


Рис 8. Вывод информации о найденной ошибке

pprint('Пирамидальная сортировка') 'Ошибка №5'

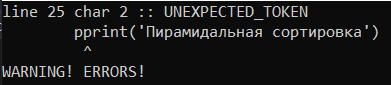


Рис 9. Вывод информации о найденной ошибке

1. **Вывод**

Был разработан лексический анализатор подмножества языка Python. Получены знания о принципах работы, обязанностях и особенностях построения лексических анализаторов языков программирования, описанных в отчёте лабораторной работы.

**Приложение. Код программы**

using System.Collections.Generic;

namespace ConsoleApp1

{

public class Token

{

public TokenTypes TokenType { get; set; }

public string Value { get; set; }

public int CodeLineNumber { get; set; }

public string Group { get; set; }

public int Length { get; set; }

public int CodeLineIndex { get; set; }

public enum TokenTypes

{

CLOSING\_SQUARE\_BRACKET,

COLON,

UNKNOWN,

OPENING\_CURLY\_BRACKET,

MULTIPLICATION,

COMMENT,

STRING\_CONST,

OPENING\_ROUND\_BRACKET,

FLOAT\_NUM,

MINUS,

CLOSING\_CURLY\_BRACKET,

INT\_NUM,

ID,

DIVISION,

DOT,

COMMA,

ASSIGN,

OPENING\_SQUARE\_BRACKET,

OR,

MODULE,

CLOSING\_ROUND\_BRACKET,

PLUS,

GREATER,

AND,

EQUAL,

NOT,

GREATER\_OR\_EQUAL,

FUNCTION\_DEFINITION,

ELIF,

BUILT\_IN\_FUNCTION,

RAISE,

IF,

IMPORT,

ELSE,

LOWER,

LOWER\_OR\_EQUAL,

WHILE,

NOT\_EQUAL,

FOR,

IN

}

public bool IsOperation

{

get => SimpleOperators.ContainsValue(TokenType) | BlockOpeningOperators.ContainsValue(TokenType);

}

public bool IsReservedIdToken

{

get => ReservedIDs.ContainsValue(TokenType);

}

public bool IsIf

{

get => TokenType == TokenTypes.IF;

}

public bool IsBlockOpeningOperation

{

get => BlockOpeningOperators.ContainsValue(TokenType);

}

public bool IsElse

{

get => TokenType == TokenTypes.ELSE;

}

public bool IsElif

{

get => TokenType == TokenTypes.ELIF;

}

public bool IsClosingBracket

{

get => this.TokenType == TokenTypes.CLOSING\_ROUND\_BRACKET;

}

public bool IsOpeningBracket

{

get => this.TokenType == TokenTypes.OPENING\_ROUND\_BRACKET;

}

public string DescriptionString

{

get

{

if (this.IsReservedIdToken && !this.IsOperation)

return $"Reserved keyword {this.TokenType}";

if (this.IsOperation)

return $"Operation {this.TokenType}";

if (this.IsConstant)

return $"{this.TokenType} constant";

if (this.TokenType == TokenTypes.COMMENT)

return $"is # comment";

return $"is {this.TokenType}";

}

}

public bool IsConstant

{

get => this.TokenType == TokenTypes.INT\_NUM

|| this.TokenType == TokenTypes.STRING\_CONST

|| this.TokenType == TokenTypes.FLOAT\_NUM;

}

public static Dictionary<string, TokenTypes> BlockOpeningOperators = new Dictionary<string, TokenTypes>()

{

["while"] = TokenTypes.WHILE,

["elif"] = TokenTypes.ELIF,

["import"] = TokenTypes.IMPORT,

["def"] = TokenTypes.FUNCTION\_DEFINITION,

["else"] = TokenTypes.ELSE,

["and"] = TokenTypes.AND,

["if"] = TokenTypes.IF,

["or"] = TokenTypes.OR,

["raise"] = TokenTypes.RAISE,

["not"] = TokenTypes.NOT,

["for"] = TokenTypes.FOR,

["in"] = TokenTypes.IN

};

public static Dictionary<string, TokenTypes> SimpleOperators = new Dictionary<string, TokenTypes>()

{

["/"] = TokenTypes.DIVISION,

["="] = TokenTypes.ASSIGN,

[":"] = TokenTypes.COLON,

["=="] = TokenTypes.EQUAL,

["{"] = TokenTypes.OPENING\_CURLY\_BRACKET,

["%"] = TokenTypes.MODULE,

[">"] = TokenTypes.GREATER,

["+"] = TokenTypes.PLUS,

["["] = TokenTypes.OPENING\_SQUARE\_BRACKET,

[")"] = TokenTypes.CLOSING\_ROUND\_BRACKET,

["-"] = TokenTypes.MINUS,

["!="] = TokenTypes.NOT\_EQUAL,

["]"] = TokenTypes.CLOSING\_SQUARE\_BRACKET,

["("] = TokenTypes.OPENING\_ROUND\_BRACKET,

["<="] = TokenTypes.LOWER\_OR\_EQUAL,

["\*"] = TokenTypes.MULTIPLICATION,

["."] = TokenTypes.DOT,

[","] = TokenTypes.COMMA,

[">="] = TokenTypes.GREATER\_OR\_EQUAL,

["}"] = TokenTypes.CLOSING\_CURLY\_BRACKET,

["<"] = TokenTypes.LOWER

};

public static Dictionary<string, TokenTypes> ReservedIDs = new Dictionary<string, TokenTypes>()

{

["type"] = TokenTypes.BUILT\_IN\_FUNCTION,

["print"] = TokenTypes.BUILT\_IN\_FUNCTION,

["min"] = TokenTypes.BUILT\_IN\_FUNCTION,

["in"] = TokenTypes.IN,

["input"] = TokenTypes.BUILT\_IN\_FUNCTION,

["or"] = TokenTypes.OR,

["range"] = TokenTypes.BUILT\_IN\_FUNCTION,

["if"] = TokenTypes.IF,

["else"] = TokenTypes.ELSE,

["import"] = TokenTypes.IMPORT,

["abs"] = TokenTypes.BUILT\_IN\_FUNCTION,

["elif"] = TokenTypes.ELIF,

["max"] = TokenTypes.BUILT\_IN\_FUNCTION,

["not"] = TokenTypes.NOT,

["def"] = TokenTypes.FUNCTION\_DEFINITION,

["int"] = TokenTypes.BUILT\_IN\_FUNCTION,

["raise"] = TokenTypes.RAISE,

["while"] = TokenTypes.WHILE,

["float"] = TokenTypes.BUILT\_IN\_FUNCTION,

["and"] = TokenTypes.AND,

["for"] = TokenTypes.FOR

};

public override string ToString()

{

return $"{TokenType}: {Value}";

}

}

}

using System.Collections.Generic;

using System.Text;

using System.Text.RegularExpressions;

using static ConsoleApp1.Token;

namespace ConsoleApp1

{

public class LexicalAnalizer

{

private List<Construction> \_constructions = new List<Construction>();

private const string COMMENT\_REGEX\_GROUP = "Comment";

private const string STRING\_REGEX\_GROUP = "String";

private const string ID\_REGEX\_GROUP = "ID";

private const string FLOAT\_REGEX\_GROUP = "Float";

private const string OTHER\_REGEX\_GROUP = "Other";

private const string INTEGER\_REGEX\_GROUP = "Integer";

private Regex \_regex = new Regex(

@"\s\*(?:(?<Comment>#.\*)|(?<String>[\""'].\*[\""'])"

+ @"|(?<Float>[+-]\*[0-9]+\.[0-9]\*)|(?<Integer>[+-]\*\d+)"

+ @"|(?<Operator>[+\-\/\*<=>!%(){},\[\]:]+)"

+ @"|(?<ID>\w+)|(?<Other>.+\s?))",

RegexOptions.Compiled | RegexOptions.IgnoreCase

);

private const string OPERATOR\_REGEX\_GROUP = "Operator";

public Construction AnaliseLine(string codeLine, int lineNumber)

{

var trimedLine = codeLine.TrimStart('\t');

int spaces = codeLine.Length - trimedLine.Length;

var (tokens, errors) = ParseLine(trimedLine, lineNumber);

return new Construction()

{

Tokens = tokens,

Errors = errors,

Indentation = spaces

};

}

public void AnaliseLines(IEnumerable<string> codeLines)

{

int lineNumber = 0;

foreach (string line in codeLines)

{

var analiseResult = AnaliseLine(line, lineNumber);

\_constructions.Add(analiseResult);

lineNumber++;

}

}

public (List<Token> tokens, List<LexicalError> errors) ParseLine(string codeLine, int lineNumber)

{

List<Token> tokens = new List<Token>();

List<LexicalError> errors = new List<LexicalError>();

MatchCollection matches = \_regex.Matches(codeLine);

string[] groupNames = \_regex.GetGroupNames();

foreach (Match match in matches)

{

GroupCollection groups = match.Groups;

for (int i = 1; i < groupNames.Length; i++)

{

if (groups[groupNames[i]].Success)

{

string trimmedValue = groups[groupNames[i]].Value.Trim(' ');

if (trimmedValue.Length == 0)

break;

TokenTypes type = GetTokenType(groupNames[i], trimmedValue);

if (type == TokenTypes.UNKNOWN)

{

LexicalError error = new LexicalError()

{

CodeLineNumber = lineNumber,

Value = groups[groupNames[i]].Value,

IndexInCodeLine = match.Index,

Length = match.Length

};

error.CreateAndSetDescription(codeLine);

errors.Add(error);

}

tokens.Add(

new Token

{

Value = groups[groupNames[i]].Value,

Group = groupNames[i],

CodeLineNumber = lineNumber,

CodeLineIndex = match.Index,

Length = match.Length,

TokenType = type

}

);

}

}

}

return (tokens, errors);

}

public static TokenTypes GetTokenType(string matchGroup, string value)

{

switch (matchGroup)

{

case COMMENT\_REGEX\_GROUP:

return TokenTypes.COMMENT;

case FLOAT\_REGEX\_GROUP:

return TokenTypes.FLOAT\_NUM;

case STRING\_REGEX\_GROUP:

return TokenTypes.STRING\_CONST;

case ID\_REGEX\_GROUP:

return GetIDTokenType(value);

case INTEGER\_REGEX\_GROUP:

return TokenTypes.INT\_NUM;

case OPERATOR\_REGEX\_GROUP:

return GetOperatorTokenType(value);

case OTHER\_REGEX\_GROUP:

default:

return TokenTypes.UNKNOWN;

}

}

public static TokenTypes GetIDTokenType(string value)

{

if (ReservedIDs.TryGetValue(value, out TokenTypes tokenType)

|| SimpleOperators.TryGetValue(value, out tokenType)

|| BlockOpeningOperators.TryGetValue(value, out tokenType))

{

return tokenType;

}

return TokenTypes.ID;

}

public static TokenTypes GetOperatorTokenType(string value)

{

if (SimpleOperators.TryGetValue(value, out TokenTypes tokenType) || BlockOpeningOperators.TryGetValue(value, out tokenType))

{

return tokenType;

}

return TokenTypes.UNKNOWN;

}

public class Construction

{

public List<Token> Tokens { get; set; }

public List<LexicalError> Errors { get; set; }

public bool HasErrors { get => Errors.Count > 0; }

public int Indentation { get; set; }

}

public class LexicalError

{

public enum ErrorTypes { UNEXPECTED\_TOKEN, UNDEFINED\_FUNCTION }

public ErrorTypes ErrorType { get; set; }

public string Value { get; set; }

public int CodeLineNumber { get; set; }

public int IndexInCodeLine { get; set; }

public int Length { get; set; }

protected string \_decription = "Unexpected token";

public string Description => \_decription;

public void CreateAndSetDescription(string codeLine)

{

StringBuilder stringBuilder = new StringBuilder(codeLine);

stringBuilder.AppendLine();

stringBuilder.Append(new string(' ', this.IndexInCodeLine));

stringBuilder.Append('^');

this.\_decription = stringBuilder.ToString();

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using static ConsoleApp1.Token;

using static ConsoleApp1.LexicalAnalizer;

using ConsoleTables;

namespace ConsoleApp1

{

class Program

{

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 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 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.IsConstant)

constants.TryAdd(token.Value, token);

else if (token.IsOperation)

operators.TryAdd(token.Value, token);

else if (token.TokenType != TokenTypes.UNKNOWN)

variables.TryAdd(token.Value, token);

}

if (construction.HasErrors)

{

foreach (LexicalError error in construction.Errors)

{

Console.WriteLine($"line {error.CodeLineNumber + 1} char {error.IndexInCodeLine + 1} :: {error.ErrorType}");

Console.WriteLine(error.Description);

}

Console.WriteLine("WARNING! ERRORS!");

Console.Read();

Environment.Exit(1);

}

SyntaxAnalizer.ExpressionNode node = null;

bool isElifElseNode = false;

node = sa.Analyse(construction.Tokens, out bool 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;

}

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

);

}

else 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

);

}

}

previousLineIndentation = construction.Indentation;

lineNumber++;

if (errors.Any())

{

foreach (LexicalError error in errors)

{

Console.WriteLine($"line {error.CodeLineNumber + 1} char {error.IndexInCodeLine + 1} :: {error.ErrorType}");

Console.WriteLine(error.Description);

}

Console.WriteLine("WARNING! ERRORS!");

}

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);

}

Console.WriteLine("\n \t\t CONSTANTS");

PrintTokensDictionary(constants);

Console.WriteLine("\n \t\t VARIABLES");

PrintTokensDictionary(variables);

Console.WriteLine("\n \t\t KEYWORDS");

PrintTokensDictionary(keywords);

Console.WriteLine("\n \t\t OPERATORS");

PrintTokensDictionary(operators);

}

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 Main(string[] args)

{

Console.OutputEncoding = System.Text.Encoding.UTF8;

string FILENAME = @"D:/6 SEM/MTran-Labs/Lab2/Lab2/test.py";

IEnumerable<string> codeLines = System.IO.File.ReadLines(FILENAME);

try

{

DoTheJob(codeLines);

}

catch (InvalidOperationException e)

{

Console.WriteLine($"SYNTAX ERROR {e.Message}");

Console.WriteLine("block opening element has nothing in its block!");

}

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()));

}

Console.Read();

}

}

}