Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №2

По теме “Лексический анализатор.”

Выполнил:

студент гр. 753504

Осипик И. Ф.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2020

1. **Постановка задачи**

Разработка лексического анализатора подмножества языка программирования, определённого в лабораторной работе №1.

Программа анализа определяет лексические правила и выполняет перевод потока символов программ лабораторной работы №1 в поток лексем (токенов).

Анализируем подмножества языка программирования Python.   
Для написания анализатора использован язык программирования С#.

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.

Обычно взаимодействие реализуется как вызов лексического анализатора синтаксическим анализатором. Этот вызов, представленный как команда getNextToken, заставляет лексический анализатор читать символы из входного потока, пока он не сможет идентифицировать очередную лексему и вернуть синтаксическому анализатору корректный 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.

Если исходная программа использует макропрепроцессор, то раскрытие макросов также может выполняться лексическим анализатором.

# **3. 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.

После выделения из строки файла все токены, в соответствии с группой совпадения, которой они были отловлены, проходят этап классификации, в ходе которой будет установлен тип каждого токена и одновременно, некоторая возможная на данном этапе, проверка допустимости вхождения такого токена в код. С помощью словарей констант токены, попавшие в группу word, проверяются на совпадение с имеющимися в выбранном подмножестве языка операторами и встроенными функциями. Если совпадение найдено, им присваивается либо тип BUILT\_IN\_FUNCTION, либо тип, соответствующий какому-то оператору.

Для представления в программе токена как объекта используется класс Token, содержащий в полях информацию о месте токена в коде (номер строки, номер первого символа в строке), его типе и собственно его строковое значение. Также в статических полях класса – словари зарезервированных операторов, функций подможества языка и тип-перечисление TokenTypes, представляющий унифицированную идентификацию токена по принадлежности его к какой-то группе (См листинг 1).

***Листинг 1. Класс Token***

public class Token

{

public string Value { get; set; }

public string Group { get; set; }

public TokenTypes TokenType { get; set; }

public int CodeLineNumber { get; set; }

public int CodeLineIndex { get; set; }

public int Length { get; set; }

public bool IsReservedIdToken

{

get => ReservedIDs.ContainsValue(TokenType);

}

public bool IsOperation

{

get => ExistingOperators.ContainsValue(TokenType);

}

public bool IsConstant

{

get => this.TokenType == TokenTypes.STRING\_CONST

|| this.TokenType == TokenTypes.INT\_NUM

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

}

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

{

["and"] = TokenTypes.AND,

["or"] = TokenTypes.OR,

["not"] = TokenTypes.NOT,

["for"] = TokenTypes.FOR,

["in"] = TokenTypes.IN,

["while"] = TokenTypes.WHILE,

["if"] = TokenTypes.IF,

["elif"] = TokenTypes.ELIF,

["else"] = TokenTypes.ELSE,

["raise"] = TokenTypes.RAISE,

["import"] = TokenTypes.IMPORT,

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

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

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

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

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

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

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

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

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

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

};

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

{

["="] = TokenTypes.ASSIGN,

["+"] = TokenTypes.PLUS,

["-"] = TokenTypes.MINUS,

["/"] = TokenTypes.DIVISION,

["\*"] = TokenTypes.MULTIPLICATION,

["%"] = TokenTypes.MODULE,

[":"] = TokenTypes.COLON,

["<"] = TokenTypes.LOWER,

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

[">"] = TokenTypes.GREATER,

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

["=="] = TokenTypes.EQUAL,

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

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

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

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

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

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

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

["."] = TokenTypes.DOT

};

public enum TokenTypes

{

UNKNOWN,

COMMENT,

STRING\_CONST,

FLOAT\_NUM,

INT\_NUM,

ID,

DOT,

COLON,

OPENING\_SQUARE\_BRACKET,

CLOSING\_SQUARE\_BRACKET,

OPENING\_ROUND\_BRACKET,

CLOSING\_ROUND\_BRACKET,

OPENING\_CURLY\_BRACKET,

CLOSING\_CURLY\_BRACKET,

AND,

OR,

NOT,

ASSIGN,

PLUS,

MINUS,

DIVISION,

MULTIPLICATION,

MODULE,

GREATER,

GREATER\_OR\_EQUAL,

LOWER,

LOWER\_OR\_EQUAL,

EQUAL,

NOT\_EQUAL,

FOR,

IN,

WHILE,

IF,

ELIF,

ELSE,

BUILT\_IN\_FUNCTION,

RAISE,

IMPORT,

FUNCTION\_DEFINITION

}

}

***Конец листинга 1.***

В случае, если для токена не найдено ни одно совпадение, то, если он ранее был классифицирован как допустимый ID, он им и остается. Если же токен классифицирован как неизвестный, то в список ошибок строки добавляется ошибка, говорящая о неизвестном токене. Для лексических ошибок предусмотрен тип LexicalAnalizer.LexicalError, объекты которого содержат информацию о местонахождении ошибки и строку информации. Метод CreateAndSetDescription позволяет для ошибки создать строку, позволяющую, при показе ее пользователю в консоли, наглядно указать начало ошибки. Он принимает строку кода, в которой была обнаружена ошибка.

Далее, токены для каждой строки группируются в объект класса Construction, представляющий собой простой способ объединить для последующего использования список токенов строки, уровень вложенности строки, и список лексических ошибок в строке.

После объединения токенов в объект конструкции проводится простой и неполный анализ синтаксического характера. Проверяются слова, которые в анализируемом коде использованы как операторы for, while и им подобные. Если они классифицированы не как допустимые в таком контексте, то это считается ошибкой и также выносится в общий список ошибок. Также, если в коде встречается ID, который пытаются вызвать как функцию, а ранее объявления функции с таким именем не было, вхождение вызова функции считается ошибкой.

Найденные токены разбиваются на логические словари констант, переменных, операторов и затем выводятся в виде таблиц с помощью   
Nuget-пакета ConsoleTables.  
**4. Примеры работы программы**

Результаты работы программы для кода из Листинга 2 приведены на рисунках 1 и 2.

***Листинг 2. Исследуемый код***

fib1 = 1

fib2 = 1

n = input( 'Number in fibonacchi: ' )

n = int(n)

i = 0

while i < n - 2:

fib\_sum = fib1 + fib2

fib1 = fib2

fib2 = fib\_sum

i = i + 1

print( fib2 )

***Конец листинга 2.***

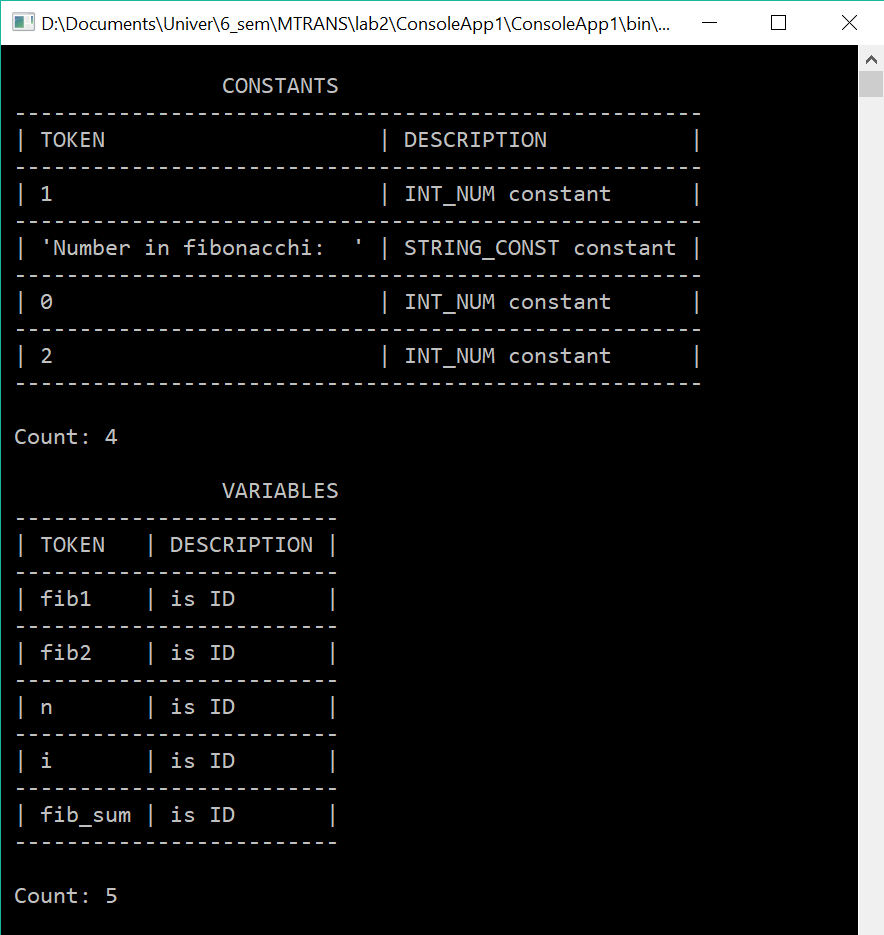


Рис 1. Первая часть вывода

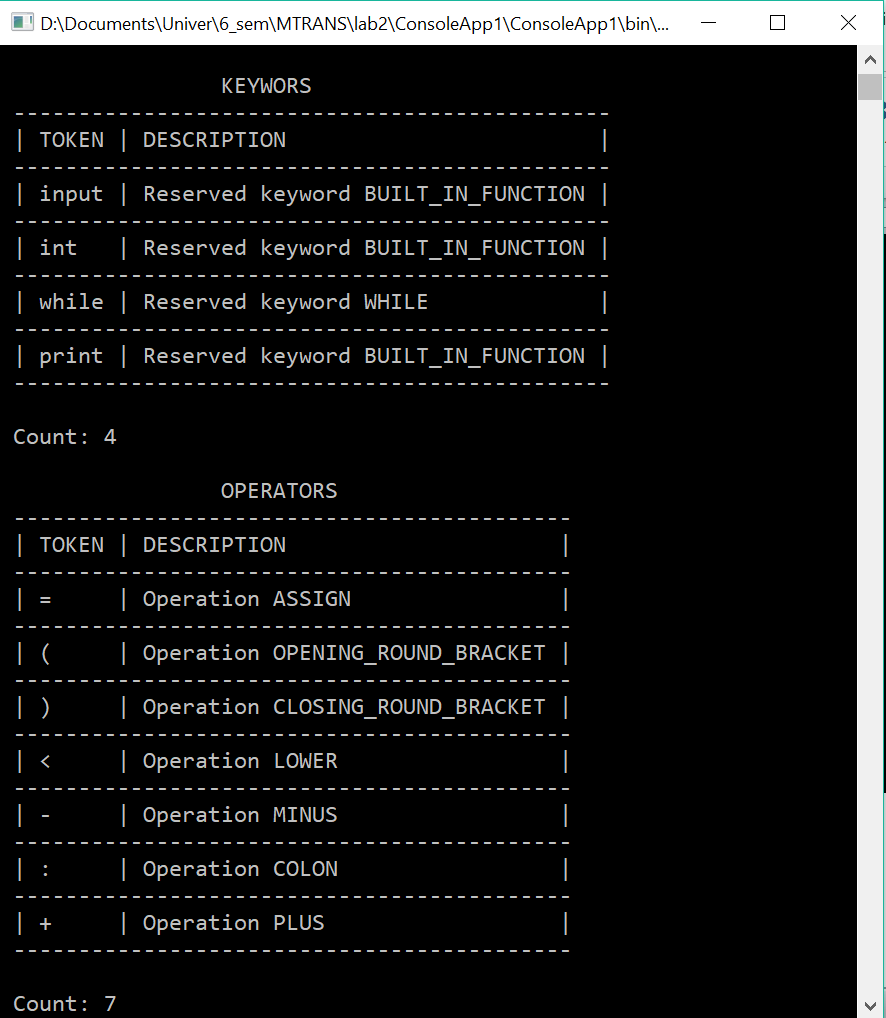


Рис. 2 – вторая часть вывода

Внесем ошибки в код. Код с ошибками и комментариями, указывающими на них приведен в листинге 3.

***Листинг 3. Код с ошибками***

# ошибка 1, === - так нельзя

fib1 === 1

fib2 = 1

# ошибка 2, нет такого слова inpput

n = inpput( 'Number in fibonacchi: ' )

n = int(n)

i = 0

# ошибка 3, в питоне нет точки с запятой

if i > 2;

print(i)

# ошибка 4 - $ тут не к месту

whi$le i < n - 2

fib\_sum = fib1 + fib2

fib1 = fib2

fib2 = fib\_sum

i = i + 1

print( fib2 )

***Конец листинга 3***



Рис 2. Вывод информации о найденных ошибках

Внесём ошибки в код. Введём несколько операндов подряд, что является ошибкой.

***Листинг 4. Код ошибками***

fib1 = 1

fib2 = 1

n = input( 'Number in fibonacchi: ' )

n = int(n)

i = 0

while i < n -+ 2:

fib\_sum = fib1 + fib2

fib1 = fib2

fib2 = fib\_sum

i = i + 1

print( fib2 )

***Конец листинга 4***

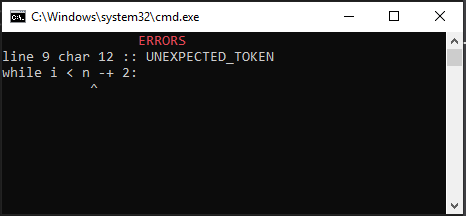


Рис 2. Вывод информации о найденной ошибке

**5. 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;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

using static ConsoleApp1.Token;

namespace ConsoleApp1

{

public class LexicalAnalizer

{

private List<Token> \_tokens = new List<Token>();

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

private List<LexicalError> \_error = new List<LexicalError>();

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

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

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

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

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

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

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

private Regex \_regex = new Regex(

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

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

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

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

RegexOptions.Compiled | RegexOptions.IgnoreCase

);

public void AnaliseLines(IEnumerable<string> codeLines)

{

int lineNumber = 0;

foreach (string line in codeLines)

{

var analiseResult = AnaliseLine(line, lineNumber);

\_constructions.Add(analiseResult);

lineNumber++;

}

}

public Construction AnaliseLine(string codeLine, int lineNumber)

{

var trimedLine = codeLine.Trim(' ');

int spaces = codeLine.Length - trimedLine.Length;

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

// if lexemes[0] == "for" && lexemes[2] == "in" => return For

return new Construction()

{

Tokens = tokens,

Errors = errors,

Indentation = spaces

};

}

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 STRING\_REGEX\_GROUP:

return TokenTypes.STRING\_CONST;

case FLOAT\_REGEX\_GROUP:

return TokenTypes.FLOAT\_NUM;

case INTEGER\_REGEX\_GROUP:

return TokenTypes.INT\_NUM;

case ID\_REGEX\_GROUP:

return GetIDTokenType(value);

case OPERATOR\_REGEX\_GROUP:

return GetOperatorTokenType(value);

case OTHER\_REGEX\_GROUP:

default:

return TokenTypes.UNKNOWN;

}

}

public static TokenTypes GetOperatorTokenType(string value)

{

TokenTypes tokenType;

if (ExistingOperators.TryGetValue(value, out tokenType))

{

return tokenType;

}

return TokenTypes.UNKNOWN;

}

public static TokenTypes GetIDTokenType(string value)

{

TokenTypes tokenType;

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

{

return tokenType;

}

return TokenTypes.ID;

}

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();

}

}

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 bool IsBlockOpeningConstruction()

{

return this.Tokens.Last().TokenType == TokenTypes.COLON;

}

public bool IncludesAssignmentConstruction()

{

foreach (var token in Tokens)

{

if (token.TokenType == TokenTypes.ASSIGN)

return true;

}

return false;

}

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1

{

public class Token

{

public string Value { get; set; }

public string Group { get; set; }

public TokenTypes TokenType { get; set; }

public int CodeLineNumber { get; set; }

public int CodeLineIndex { get; set; }

public int Length { get; set; }

public bool IsReservedIdToken

{

get => ReservedIDs.ContainsValue(TokenType);

}

public bool IsOperation

{

get => ExistingOperators.ContainsValue(TokenType);

}

public bool IsConstant

{

get => this.TokenType == TokenTypes.STRING\_CONST

|| this.TokenType == TokenTypes.INT\_NUM

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

}

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

{

["and"] = TokenTypes.AND,

["or"] = TokenTypes.OR,

["not"] = TokenTypes.NOT,

["for"] = TokenTypes.FOR,

["in"] = TokenTypes.IN,

["while"] = TokenTypes.WHILE,

["if"] = TokenTypes.IF,

["elif"] = TokenTypes.ELIF,

["else"] = TokenTypes.ELSE,

["raise"] = TokenTypes.RAISE,

["import"] = TokenTypes.IMPORT,

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

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

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

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

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

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

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

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

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

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

};

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

{

["="] = TokenTypes.ASSIGN,

["+"] = TokenTypes.PLUS,

["-"] = TokenTypes.MINUS,

["/"] = TokenTypes.DIVISION,

["\*"] = TokenTypes.MULTIPLICATION,

["%"] = TokenTypes.MODULE,

[":"] = TokenTypes.COLON,

["<"] = TokenTypes.LOWER,

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

[">"] = TokenTypes.GREATER,

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

["=="] = TokenTypes.EQUAL,

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

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

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

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

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

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

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

["."] = TokenTypes.DOT

};

public enum TokenTypes

{

UNKNOWN,

COMMENT,

STRING\_CONST,

FLOAT\_NUM,

INT\_NUM,

ID,

DOT,

COLON,

OPENING\_SQUARE\_BRACKET,

CLOSING\_SQUARE\_BRACKET,

OPENING\_ROUND\_BRACKET,

CLOSING\_ROUND\_BRACKET,

OPENING\_CURLY\_BRACKET,

CLOSING\_CURLY\_BRACKET,

AND,

OR,

NOT,

ASSIGN,

PLUS,

MINUS,

DIVISION,

MULTIPLICATION,

MODULE,

GREATER,

GREATER\_OR\_EQUAL,

LOWER,

LOWER\_OR\_EQUAL,

EQUAL,

NOT\_EQUAL,

FOR,

IN,

WHILE,

IF,

ELIF,

ELSE,

BUILT\_IN\_FUNCTION,

RAISE,

IMPORT,

FUNCTION\_DEFINITION

}

}

}

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 void Main(string[] args)

{

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

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>();

string FILENAME = @"D:/Documents/Univer/6\_sem/MTRANS/lab2/ConsoleApp1/test.py";

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

// running lexical analysis

LexicalAnalizer lexicalAnalizer = new LexicalAnalizer();

int lineNumber = 0;

foreach (string line in codeLines)

{

Construction construction = lexicalAnalizer.AnaliseLine(line, lineNumber);

if (construction.Tokens.Count == 0)

continue;

if (construction.Tokens.Last().TokenType == TokenTypes.COLON)

{

Token firstToken = construction.Tokens.First();

TokenTypes firstTokenType = firstToken.TokenType;

if (firstTokenType != TokenTypes.WHILE && firstTokenType != TokenTypes.FOR && firstTokenType != TokenTypes.FUNCTION\_DEFINITION)

{

LexicalError error = new LexicalError()

{

Value = construction.Tokens.First().Value,

CodeLineNumber = lineNumber,

IndexInCodeLine = 0,

ErrorType = LexicalError.ErrorTypes.UNDEFINED\_FUNCTION

};

error.CreateAndSetDescription(line);

errors.Add(error);

}

else if (firstTokenType == TokenTypes.FUNCTION\_DEFINITION) {

Token secondToken = construction.Tokens[1];

if (secondToken.TokenType == TokenTypes.ID) {

variables.Add(secondToken.Value, secondToken);

}

}

}

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)

{

if (i < construction.Tokens.Count - 1 && construction.Tokens[i + 1].TokenType == TokenTypes.OPENING\_ROUND\_BRACKET)

{

if (!variables.ContainsKey(construction.Tokens[i + 1].Value))

{

LexicalError error = new LexicalError()

{

Value = token.Value,

CodeLineNumber = lineNumber,

IndexInCodeLine = line.IndexOf(token.Value),

ErrorType = LexicalError.ErrorTypes.UNDEFINED\_FUNCTION

};

error.CreateAndSetDescription(line);

errors.Add(error);

}

}

variables.TryAdd(token.Value, token);

}

}

if (construction.HasErrors)

errors.AddRange(construction.Errors);

lineNumber++;

}

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 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);

Console.Read();

}

}

}