Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №4

По теме “Семантический анализатор.”

Выполнил:

студент гр. 953501

Кондрашов И.Д.

Проверил:

Ассистент кафедры информатики

Шиманский В. В.

Минск 2022

Содержание

1. **Постановка задачи**
2. **Теория**
3. **Пример работы программы**
4. **Вывод**

**Приложение. Код программы**

1. **Постановка задачи**

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций.

Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насильственное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное.

В большинстве языков программирования имеет место неявное изменение типов (иногда называемое приведением типов(coercion)). Реже встречаются языки, подобные Ada, в которых большинство изменений типов должно быть явным.

В языках со статическими типами, например С, все типы известны во время компиляции, и это относится к типам выражений, идентификаторам и литералам. При этом неважно, насколько сложным является выражение: его тип может определяться во время компиляции за определенное количество шагов, исходя из типов его составляющих. Фактически, это позволяет производить контроль типов во время компиляции и находить заранее (в процессе компиляции, а не во время выполнения программы!) многие программные ошибки.

1. **Теория**

Семантический анализ является центральной фазой трансляции, связывающей 2 ее логические части: анализ исходной программы и синтез объектной программы. На этапе семантического анализа обрабатываются программные конструкции, распознанные синтаксическим анализатором.

Основные функции семантического анализатора:

1. *Заполнение таблиц имен*.
2. *Выделение неявно заданной информации*.
3. *Обнаружение ошибок.*
4. *Выполнение некоторых операций программы*.
5. *Формирование внутренней формы программы*.

Формирование внутренней формы исходной программы семантический анализатор (СА) осуществляет не для всей программы в целом, а по частям, последовательно для каждой распознанной синтаксической конструкции. Поэтому СА обычно состоит из ряда процедур, каждая из которых предназначена для обработки конструкций конкретного типа, и называемых семантическими программами. Взаимодействие семантических программ производится путем обмена данными через информационную таблицу (таблицу имен).

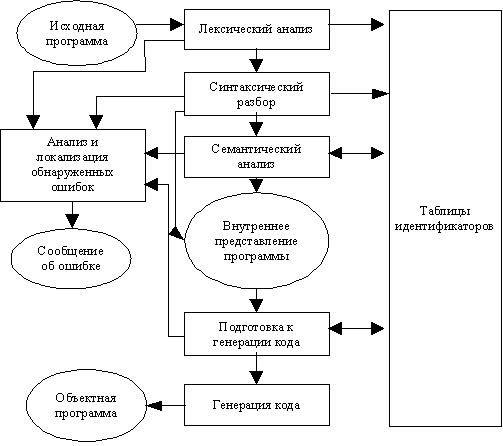


Рис 1. Схема работы интерпретатора

Цель идентификации идентификаторов – определить тип использующего вхождения идентификатора. Эта задача может быть полностью или частично решена в фазе синтаксического анализа. Все зависит от того, может ли использующее вхождение идентификатора встретиться в программе до определяющего вхождения, или нет.

Если все определяющие вхождения идентификаторов должны быть расположены текстуально перед использующими вхождениями, то можно выполнить идентификацию на фазе синтаксического анализа. Если же нет, то в фазе синтаксического анализа мы можем обработать определяющие вхождения идентификаторов и только на следующем просмотре текста программы выполнить собственно идентификацию.

Вне зависимости от того, на каком просмотре будет выполняться идентификация идентификаторов, при обработке определяющего вхождения идентификатора необходимо запомнить информацию о типе этого идентификатора.

Если контроль типов осуществляется во время трансляции программы, то мы говорим о статическом контроле типов, в противном случае, т. е. если контроль типов производится во время исполнения объектной программы, мы говорим о динамическом контроле типов. В принципе, контроль типов всегда может выполняться динамически, если в объектном коде вместе со значением будет размещаться и тип этого значения.

Необходимой частью контроля типов является проверка эквивалентности типов (equivalence of types). Крайне необходимо, чтобы компилятор выполнял проверку эквивалентности типов быстро.

1. **Пример работы программы**

def heapify(nums, heap, root) :  
 largest = root  
 left = (2 \* root) + 1  
 right = ((2 \* root) + 2) - 'some\_text' 'Ошибка №1'

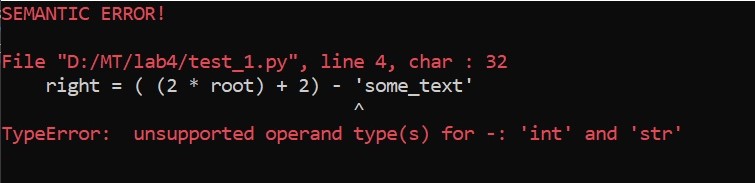


Рис 2. Вывод информации о найденной ошибке

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)  
  
random\_nums = [46, 18, 1, 35, 26]  
heap\_sort() 'Ошибка №2'

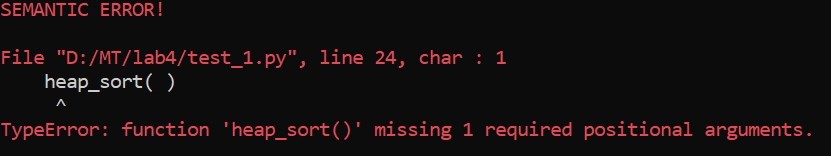


Рис 3. Вывод информации о найденной ошибке

**4. Вывод**

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций.

**Приложение. Код программы**

using System;

using System.Collections.Generic;

using System.Linq;

using static ConsoleApp1.Token;

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 (tokens.LastOrDefault()?.TokenType == Token.TokenTypes.COLON)

{

throw new SyntaxErrorException("something wrong with a keyword", firstToken.Value, firstToken.CodeLineIndex, firstToken.CodeLineNumber);

}

else 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("something wrong with brackets", 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.IsOpeningBracket)

{

this.OpenedBracketsLevel++;

root = BuildTree(tokens.Skip(1));

root.OperatorPriority++;

}

else 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.COLON)

{

root = left;

left = null;

root.Right = BuildTree(tokens.Skip(2));

}

else if (tt == Token.TokenTypes.OPENING\_ROUND\_BRACKET)

{

root = left;

left = null;

root.Type = ExpressionNode.ExpressionTypes.FUNCTION\_CALL;

root.Right = BuildTree(tokens.Skip(1));

}

else

{

root = BuildTree(tokens.Skip(1));

left.Parent = root;

}

}

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);

}

else if (token.IsClosingBracket)

{

this.OpenedBracketsLevel--;

root = BuildTree(tokens.Skip(1));

if (root != null)

root.OperatorPriority--;

}

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.BLOCK\_OPENING\_CONDITIONAL\_OPERATION:

if (node.Left != null || node.Right == null)

throw new SyntaxErrorException(

"something wrong with conditional operator",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

break;

case ExpressionNode.ExpressionTypes.BINARY\_OPERATION:

if (node.Left == null || node.Right == null)

{

throw new SyntaxErrorException(

"something wrong with binary operation",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

break;

case ExpressionNode.ExpressionTypes.OPERAND:

if (node.Left != null)

throw new SyntaxErrorException(

"unknown operator",

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

);

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 void InsertDeepLeft(ExpressionNode node)

{

ExpressionNode temp = this;

while (!(temp.Left is null))

{

temp = temp.Left;

}

temp.Left = node;

}

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 static Dictionary<TokenTypes, ExpressionTypes> TokensToExpressionTypes = new Dictionary<TokenTypes, ExpressionTypes>()

{

[TokenTypes.ELIF] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.DOT] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.FUNCTION\_DEFINITION] = ExpressionTypes.FUNCTION\_DEF,

[TokenTypes.ASSIGN] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.WHILE] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.COMMA] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.MINUS] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.IF] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.NOT] = ExpressionTypes.UNARY\_OPERATION,

[TokenTypes.FLOAT\_NUM] = ExpressionTypes.OPERAND,

[TokenTypes.ELSE] = ExpressionTypes.BLOCK\_OPENING\_OPERATION,

[TokenTypes.IN] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.FOR] = ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

[TokenTypes.ID] = ExpressionTypes.OPERAND,

[TokenTypes.PLUS] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.GREATER] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.NOT\_EQUAL] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.MODULE] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.DIVISION] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.MULTIPLICATION] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.INT\_NUM] = ExpressionTypes.OPERAND,

[TokenTypes.AND] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.OR] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.BUILT\_IN\_FUNCTION] = ExpressionTypes.OPERAND,

[TokenTypes.LOWER] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.LOWER\_OR\_EQUAL] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.COLON] = ExpressionTypes.BLOCK\_OPENING\_OPERATION,

[TokenTypes.GREATER\_OR\_EQUAL] = ExpressionTypes.BINARY\_OPERATION,

[TokenTypes.STRING\_CONST] = ExpressionTypes.OPERAND,

[TokenTypes.EQUAL] = ExpressionTypes.BINARY\_OPERATION

};

public enum ExpressionTypes

{

BINARY\_OPERATION,

UNKNOWN,

FUNCTION\_DEF,

UNARY\_OPERATION,

OPERAND,

BLOCK\_OPENING\_OPERATION,

BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

FUNCTION\_CALL

};

public override string ToString()

{

return $"({Operator})";

}

}

}

}

using System.Collections.Generic;

namespace ConsoleApp1

{

class SemanticItem

{

public string Name { get; set; }

public VarTypes VarType { get; set; }

public FunctionSpecification FSpecification { get; set; }

public struct FunctionSpecification

{

public VarTypes ReturnType { get; set; }

public int MaxArgumentsAmount { get; set; }

public int MinArgumentsAmount { get; set; }

}

public enum VarTypes

{

NUMBER\_VAR,

INTEGER\_VAR,

FLOAT\_VAR,

STRING\_VAR,

FUNCTION\_VAR,

LIST\_VAR,

BOOL\_VAR,

NONE\_VAR,

}

public static Dictionary<string, FunctionSpecification> BuiltInFunctionsReference = new Dictionary<string, FunctionSpecification>

{

["print"] = new FunctionSpecification() { ReturnType = VarTypes.NONE\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 100 },

["input"] = new FunctionSpecification() { ReturnType = VarTypes.STRING\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 1 },

["range"] = new FunctionSpecification() { ReturnType = VarTypes.INTEGER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 3 },

["type"] = new FunctionSpecification() { ReturnType = VarTypes.STRING\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 3 },

["abs"] = new FunctionSpecification() { ReturnType = VarTypes.NUMBER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 1 },

["max"] = new FunctionSpecification() { ReturnType = VarTypes.NUMBER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 100 },

["min"] = new FunctionSpecification() { ReturnType = VarTypes.NUMBER\_VAR, MinArgumentsAmount = 1, MaxArgumentsAmount = 100 },

["int"] = new FunctionSpecification() { ReturnType = VarTypes.INTEGER\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 1 },

["float"] = new FunctionSpecification() { ReturnType = VarTypes.FLOAT\_VAR, MinArgumentsAmount = 0, MaxArgumentsAmount = 1 }

};

public static Dictionary<string, VarTypes> StringVarTypes = new Dictionary<string, VarTypes>

{

["int"] = VarTypes.INTEGER\_VAR,

["float"] = VarTypes.FLOAT\_VAR,

["str"] = VarTypes.STRING\_VAR,

["bool"] = VarTypes.BOOL\_VAR,

["list"] = VarTypes.LIST\_VAR,

["None"] = VarTypes.NONE\_VAR,

["function"] = VarTypes.FUNCTION\_VAR

};

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using static ConsoleApp1.Token;

using static ConsoleApp1.LexicalAnalyzer;

namespace ConsoleApp1

{

class Program

{

static string PrintNodeWithChildren(ExpressionNode node, string indentation)

{

if (node == null)

{

return "";

}

SyntaxAnalyzer.ValidateNode(node);

StringBuilder stringBuilder = new StringBuilder();

stringBuilder.AppendLine($"{indentation} {node.Operator.Value} {node.Type}");

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<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>();

SemanticTreeList tree = new SemanticTreeList(null);

SemanticTreeList currentBlock = tree;

int lineNumber = 0;

SyntaxAnalyzer sa = new SyntaxAnalyzer();

LexicalAnalyzer la = new LexicalAnalyzer();

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)

{

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);

}

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;

}

if (node.Operator.IsElif && !currentBlock.Last().Operator.IsIf)

{

throw new SyntaxAnalyzer.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 SyntaxAnalyzer.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 SyntaxAnalyzer.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 SemanticTreeList(currentBlock);

currentBlock = currentBlock.Last().Block;

currentBlock.Indentation = construction.Indentation;

continue;

}

currentBlock.Add(node);

}

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!");

}

SemanticErrorException semanticError = RunPython.DoSemanticCheck();

Console.WriteLine("SYNTAX TREE:\n");

PrintSyntaxTree(tree);

}

static void Main(string[] args)

{

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

string FILENAME = @"D:/6 SEM/MTran-Labs/Lab4/ConsoleApp1/test.py";

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

try

{

DoTheJob(codeLines);

}

catch (SyntaxAnalyzer.SyntaxErrorException e)

{

Console.WriteLine($"File\"D:/6 SEM/MTran-Labs/Lab4/example\_1.py\", line {e.LineNumber + 1}, char {e.PositionInLine}:");

Console.WriteLine($"\t{ErrorDescription(e.PositionInLine, codeLines.ElementAt(e.LineNumber).Trim())}");

Console.WriteLine($"\t{ErrorDescription(e.PositionInLine, codeLines.ElementAt(e.LineNumber).Trim())}");

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"SyntaxError: {e.Message}");

Console.ResetColor();

}

catch (SemanticErrorException e)

{

Console.WriteLine($"SEMANTIC ERROR!\n");

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"File\"D:/6 SEM/MTran-Labs/Lab4/example\_1.py\", line {e.LineNumber}, char : {e.PositionInLine}");

Console.ResetColor();

Console.WriteLine(ErrorDescription(e.PositionInLine, e.Value));

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"{e.ErrorType}: {e.Description}");

Console.ResetColor();

}

catch (InvalidOperationException e)

{

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

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

}

Console.Read();

}

}

}