Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №3

По теме “Синтаксический 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 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.

Представление грамматических фраз исходной программы выполнить в виде дерева. Реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR-метод, метод предшествования и пр.).

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

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

<ИД1>=(<ИД2>+<ИД3>)\*<ИД4>.

По этой цепочке необходимо выполнить следующие действия:

1. <ИД3> прибавить к <ИД2>;
2. результат (1) умножить на <ИД4>;
3. результат (2) поместить в ячейку, резервированную для <ИД1>. Этой последовательности соответствует дерево, изображенное на рис. 1.

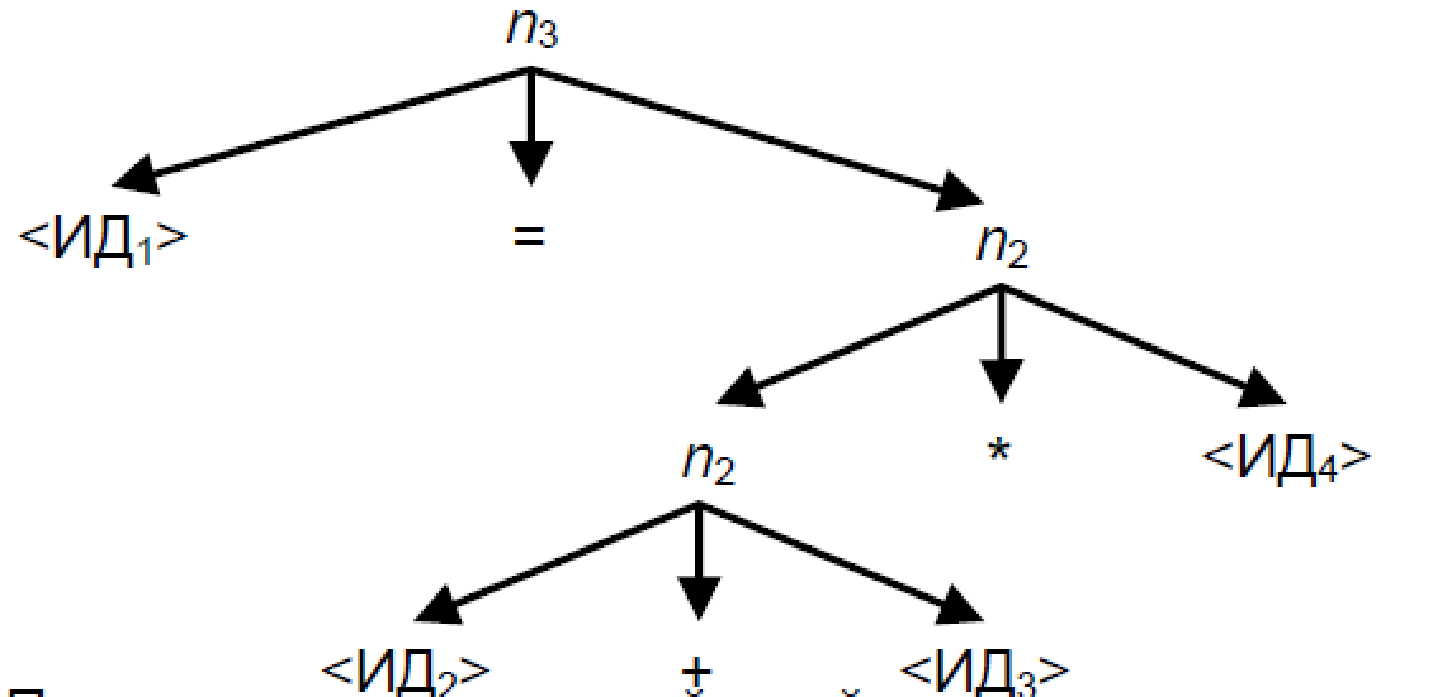


Рис. 1. – Последовательность действий при вычислении выражения

Т.е. мы имеем последовательность шагов в виде помеченного дерева.

Внутренние вершины представляют те действия, которые можно выполнять. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применять действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие, в частности, знаки «+», «\*» и «=». Скобки отсутствуют, т.к. они только определяют порядок действий.

LL- и LR- методы позволят обнаружить ошибки на самых ранних стадиях, т.е. когда разбор потока токенов от лексического анализатора в соответствии с грамматикой языка становится невозможен.

Можно использовать нисходящий (англ. top-down parser) со стартового символа, до получения требуемой последовательности токенов. Для этих целей применим метод рекурсивного спуска либо LL-анализатор. Или использовать восходящий (англ. bottom-up parser) - продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом - LR-анализатор и проч.

1. **Теория**

Синтаксический анализатор – это часть программы, преобразующей входные данные в структурированный формат; процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой, обычно применяемый совместно с лексическим анализом. Парсер выполняет синтаксический анализ текста.

В данной лабораторной работе необходимо реализовать простейший вариант синтаксического анализатора, поэтому будет достаточно использовать однопроходное абстрактное дерево.

При реализации синтаксического анализатора надо все символы, которые могут встретиться в обрабатываемом тексте разбить на группы таким образом, чтобы все символы группы вызывали одинаковую реакцию синтаксического анализатора, то есть это и есть лабораторная работа №1 по разбиению на токены. Затем необходимо выделить состояния синтаксического анализатора. Состояние определяет, какие символы в данный момент могут быть на входе синтаксического анализатора, и какова будет реакция на этот символ.

Одно состояние является начальным. Именно с него начинается работа синтаксического анализатора, и одно или несколько состояний должны быть конечными. Далее строиться таблица, которая определяет реакцию синтаксического анализатора на входные символы в зависимости от состояния. Реакция обычно заключается в смене состояния синтаксического анализатора и ещё каких-то действиях.

Если выражение записано верно, то в результате работы синтаксического анализатора должен появиться список лексем (список). Поскольку элементы списка должны иметь одинаковый тип, надо выбрать такую структуру, с помощью которой можно представить все возможные лексемы. Это можно сделать, если, например, каждый элемент списка представляет собой структуру, состоящую из двух полей: тип лексемы и её номер в списке лексем этого типа.

Часть лексем в арифметических и логических выражениях состоят из одного символа, но имена переменных и функций и константы в общем случае состоят из нескольких символов. Поэтому входные символы надо записывать во временную переменную, и когда данная лексема заканчивается, проверять, что она из себя представляет.

1. **Пример работы программы**

Пример 1. Пирамидальная сортировка

Исходный код программы на языке Python.

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)  
  
random\_nums = [0002., 0007., 0001., 0003.]  
heap\_sort(random\_nums)  
print('Heapsort')  
print(random\_nums)

Дерево программы.

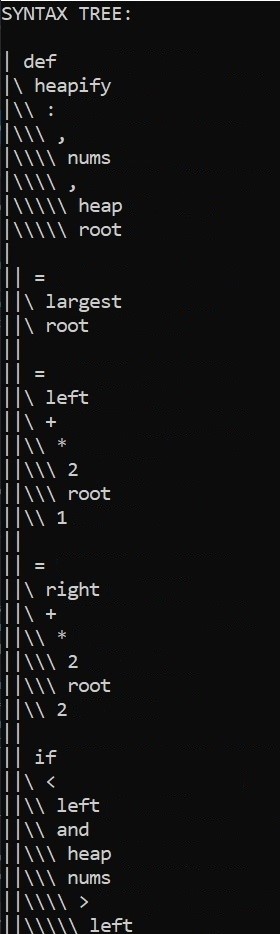


Рис 1. Первая часть дерева программы

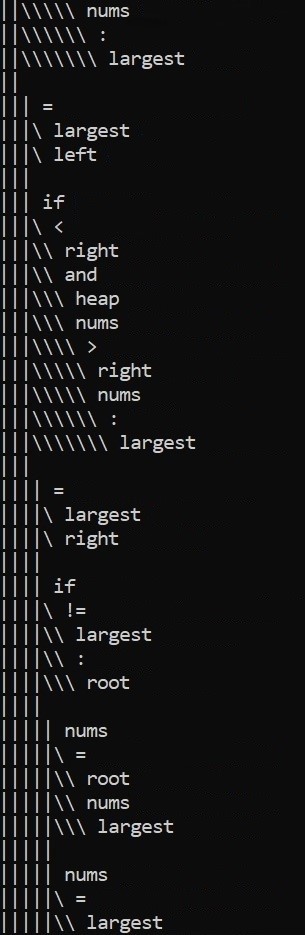


Рис 2. Вторая часть дерева программы

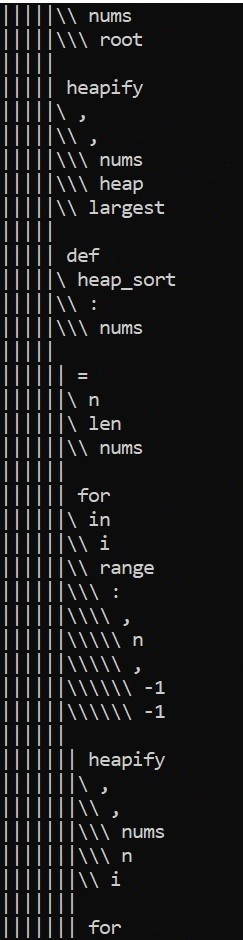


Рис 3. Третья часть дерева программы

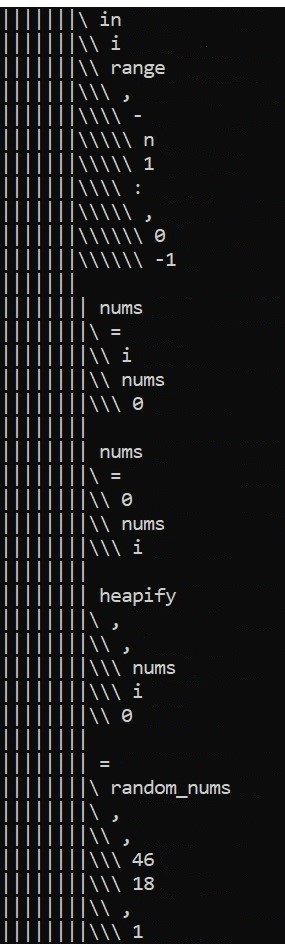


Рис 4. Четвертая часть дерева программы

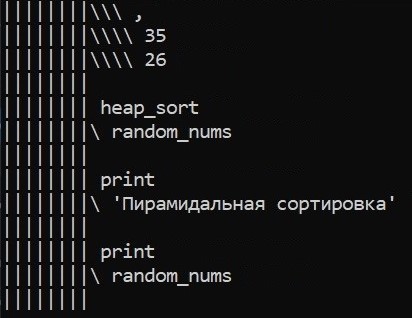


Рис 5. Пятая часть дерева программы

Все дерево в текстовом варианте:

SYNTAX TREE:

| def

|\ heapify

|\\ :

|\\\ ,

|\\\\ nums

|\\\\ ,

|\\\\\ heap

|\\\\\ root

|

|| =

||\ largest

||\ root

||

|| =

||\ left

||\ +

||\\ \*

||\\\ 2

||\\\ root

||\\ 1

||

|| =

||\ right

||\ +

||\\ \*

||\\\ 2

||\\\ root

||\\ 2

||

|| if

||\ <

||\\ left

||\\ and

||\\\ heap

||\\\ nums

||\\\\ >

||\\\\\ left

||\\\\\ nums

||\\\\\\ :

||\\\\\\\ largest

||

||| =

|||\ largest

|||\ left

|||

||| if

|||\ <

|||\\ right

|||\\ and

|||\\\ heap

|||\\\ 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

||||||\ in

||||||\\ i

||||||\\ range

||||||\\\ :

||||||\\\\ ,

||||||\\\\\ n

||||||\\\\\ ,

||||||\\\\\\ -1

||||||\\\\\\ -1

||||||

||||||| heapify

|||||||\ ,

|||||||\\ ,

|||||||\\\ nums

|||||||\\\ n

|||||||\\ i

|||||||

||||||| for

|||||||\ in

|||||||\\ i

|||||||\\ 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

||||||||\ random\_nums

||||||||

|||||||| print

||||||||\ 'Пирамидальная сортировка'

||||||||

|||||||| print

||||||||\ random\_nums

||||||||

Пример 2. Пирамидальная сортировка с ошибками

def heapify(nums, heap, root) :  
 largest = root  
 left = (2 \* root) + 1 5 'Ошибка №1'

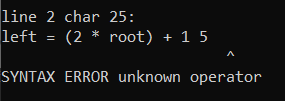


Рис 6. Вывод информации о найденной ошибке

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

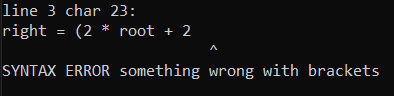


Рис 7. Вывод информации о найденной ошибке

left = (2 \* root) + 1  
right = (2 \* root) + + +2 'Ошибка №3'

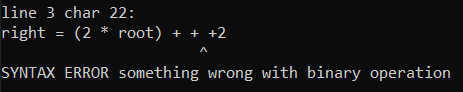


Рис 8. Вывод информации о найденной ошибке

nums[root] = nums[largest]  
nums[largest] = nums[root]  
heapify(nums, heap, largest 56 m) 'Ошибка №4'



Рис 9. Вывод информации о найденной ошибке

largest = root  
left = (2 \* root) + 1  
right = (2 \* root) + /2 'Ошибка №5'

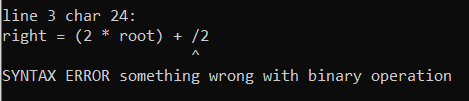


Рис 10. Вывод информации о найденной ошибке

1. **Вывод**

Сложность синтаксического анализа в доработке лексического анализатора и проверке на вложенность.

Синтаксический анализатор представляет собой вторую фазу компилятора, его основная задача состоит из создания программы, способной создавать синтаксическое дерево, состоящее из списка (list) на языке Python. Это дерево в дальнейшем необходимо для создания виртуальной машины или, другими словами, интерпретатора.

**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;

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("SYNTAX TREE:\n");

PrintSyntaxTree(tree);

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.WriteLine($"line {e.LineNumber} char {e.PositionInLine}:");

Console.WriteLine(ErrorDescription(e.PositionInLine, codeLines.ElementAt(e.LineNumber).Trim()));

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

}

Console.Read();

}

}

}