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. 753504

Осипик И.Ф.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2020

Содержание

[1. Постановка задачи: 3](#_Toc6242000)

[2. Теоретические сведения: 4](#_Toc6242001)

[3. Результат работы: 7](#_Toc6242002)

[4. Разбор ошибок 13](#_Toc6242003)

[5. Выводы 15](#_Toc6242004)

[Приложение. Код программы 16](#_Toc6242005)

## Постановка задачи:

В данной работе ставится задача исследования области синтаксического анализатора, подробному изучению существующих синтаксических анализаторов и, самое главное, написанию собственного анализатора.

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

Исследуемый код языка представлен ниже:

alist = list(54, 26, 93, 17)

def mergeSort(alist) :

print("Splitting ",alist)

if len(alist) > 1:

mid = len(alist) / 2

lefthalf = alist( :mid )

righthalf = alist( mid: )

mergeSort(lefthalf)

mergeSort(righthalf)

i=0

j=0

k=0

while i < len(lefthalf) and j < len(righthalf) :

if lefthalf(i) < righthalf(j) :

alist(k) = lefthalf(i)

i = i + 1

else :

alist(k) = righthalf(j)

j = j + 1

k = k + 1

while i < len(lefthalf) :

alist(k) = lefthalf(i)

i = i + 1

k = k + 1

while j < len(righthalf) :

alist(k) = righthalf(j)

j = j + 1

k = k + 1

print("Merging ",alist)

mergeSort(alist)

print(alist)

*Листинг. 1.1. Исходный код программы на языке Python*

## **Теоретические сведения:**

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.

Наиболее часто встречающиеся виды парсеров:

* очередь классифицированных лексем;
* абстрактное дерево;
* иерархические структуры;
* таблицы данных;

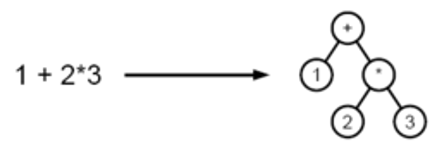
Виды парсеров по числу чтений входных данных:

* однопроходные;
* многопроходные.

Исходный код парсера может быть:

* написан программистами;
* сгенерирован специализированными утилитами;

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



*Пример 2.1. Разбор выражения с преобразованием его структуры из линейной в древовидную (в дерево иерархических зависимостей)*

Типы алгоритмов:

* Нисходящий парсер (англ. top-down parser) — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.
* Метод рекурсивного спуска
* LL-анализатор
* Восходящий парсер (англ. bottom-up parser) — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом.
* LR-анализатор
* GLR-парсер

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. Например, если на вход синтаксического анализатора пришла цифра, то он переходит в состояние «Константа», и до завершения константы (т.е. до появления пробела, знака операции, закрывающей скобки или конца строки) на вход синтаксического анализатора могут приходить только цифры и точка, если константа вещественная. Причём, естественно, точка может встречаться только один раз, что приводит к делению состояния «Константа» на два состояния: «Константа до точки», в котором точка может появиться, и «Константа после точки», в котором появление точки будет считаться ошибкой.

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, например, запись очередного символа во временную переменную, увеличение или уменьшение уровня вложенности при появлении скобок и т.д.

Обычно в программе синтаксический анализатор реализуется как бесконечный цикл (for ( ; ; ) ) с выходами в случае ошибки или при достижении конца обрабатываемой строки. Внутри цикла пишется условный блок, например, по группам символов, и в каждой части этого условного блока – свой условный блок (или оператор-переключатель) по состояниям синтаксического анализатора.

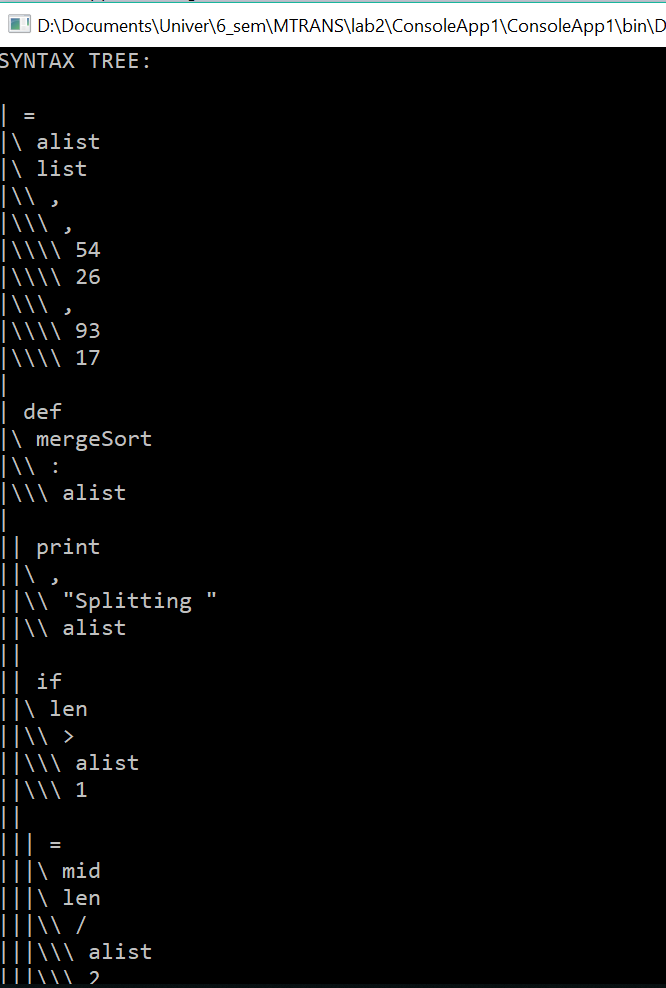
Если выражение записано верно, то в результате работы синтаксического анализатора должен появиться список лексем (список). Поскольку элементы списка должны иметь одинаковый тип, надо выбрать такую структуру, с помощью которой можно представить все возможные лексемы. Это можно сделать, если, например, 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.

Следует помнить, что одна и та же переменная может входить в выражение несколько раз, поэтому каждый раз, когда имя переменной сформировалось, надо проверять, была ли она уже в списке переменных.   
А при достижении конца обрабатываемого выражения надо проверить, что количество открывающих и закрывающих скобок было одинаковым (вложенность = 0).

## Результат работы:

В результате дерево программы имеет следующий вид Рис.3.1



*Рис. 3.1. Фрагмент дерева*

*Листинг 3.1. Все дерево*

*SYNTAX TREE:*

| =

|\ alist

|\ list

|\\ ,

|\\\ ,

|\\\\ 54

|\\\\ 26

|\\\ ,

|\\\\ 93

|\\\\ 17

|

| def

|\ mergeSort

|\\ :

|\\\ alist

|

|| print

||\ ,

||\\ "Splitting "

||\\ alist

||

|| if

||\ len

||\\ >

||\\\ alist

||\\\ 1

||

||| =

|||\ mid

|||\ len

|||\\ /

|||\\\ alist

|||\\\ 2

|||

||| =

|||\ lefthalf

|||\ alist

|||\\ :

|||\\\ mid

|||

||| =

|||\ righthalf

|||\ alist

|||\\ mid

|||

||| mergeSort

|||\ lefthalf

|||

||| mergeSort

|||\ righthalf

|||

||| =

|||\ i

|||\ 0

|||

||| =

|||\ j

|||\ 0

|||

||| =

|||\ k

|||\ 0

|||

||| while

|||\ <

|||\\ i

|||\\ len

|||\\\ and

|||\\\\ lefthalf

|||\\\\ <

|||\\\\\ j

|||\\\\\ len

|||\\\\\\ :

|||\\\\\\\ righthalf

|||

|||| if

||||\ lefthalf

||||\\ <

||||\\\ i

||||\\\ righthalf

||||\\\\ :

||||\\\\\ j

||||

||||| alist

|||||\ =

|||||\\ k

|||||\\ lefthalf

|||||\\\ i

|||||

||||| =

|||||\ i

|||||\ +

|||||\\ i

|||||\\ 1

|||||

|||| else

||||\ :

||||

||||| alist

|||||\ =

|||||\\ k

|||||\\ righthalf

|||||\\\ j

|||||

||||| =

|||||\ j

|||||\ +

|||||\\ j

|||||\\ 1

|||||

|||| =

||||\ k

||||\ +

||||\\ k

||||\\ 1

||||

||| while

|||\ <

|||\\ i

|||\\ len

|||\\\ :

|||\\\\ lefthalf

|||

|||| alist

||||\ =

||||\\ k

||||\\ lefthalf

||||\\\ i

||||

|||| =

||||\ i

||||\ +

||||\\ i

||||\\ 1

||||

|||| =

||||\ k

||||\ +

||||\\ k

||||\\ 1

||||

||| while

|||\ <

|||\\ j

|||\\ len

|||\\\ :

|||\\\\ righthalf

|||

|||| alist

||||\ =

||||\\ k

||||\\ righthalf

||||\\\ j

||||

|||| =

||||\ j

||||\ +

||||\\ j

||||\\ 1

||||

|||| =

||||\ k

||||\ +

||||\\ k

||||\\ 1

||||

|| print

||\ ,

||\\ "Merging "

||\\ alist

||

| mergeSort

|\ alist

|

| print

|\ alist

|

*Конец листинга 3.1*

## Разбор ошибок

Изменим исходный код программы, сознательно сделав несколько синтаксических ошибок:

1) Не закроем скобку:

alist = list(54, 26, 93, 17)

def mergeSort(alist) :

print("Splitting ",alist)

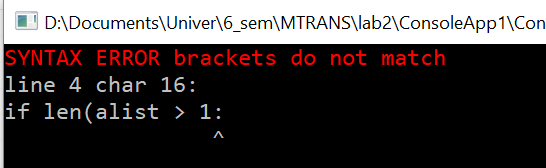
if len(alist > 1:

mid = len(alist) / 2

lefthalf = alist( :mid )

righthalf = ali

*Листинг. 4.1. Незакрытая скобка в 3й строке*



*Рис. 4.1. Ошибка отловлена*

2) Добавим число в неожиданное место без оператора:

alist = list(54, 26, 93, 17)

def mergeSort(alist) :

print("Splitting ",alist)

if len(alist > 1) :

mid = len(alist) / 2

lefthalf = alist( :mid )

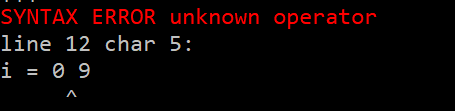
righthalf = alist( mid: )

mergeSort(lefthalf)

mergeSort(righthalf)

i = 0 9

*Листинг. 4.2. Отсутствие оператора*



*Рис. 4.2. Ошибка отловлена*

3) Введем лишние операторы «+»:

def mergeSort(alist) :

print("Splitting ",alist)

if len(alist > 1) :

mid = len(alist) / 2

lefthalf = alist( :mid )

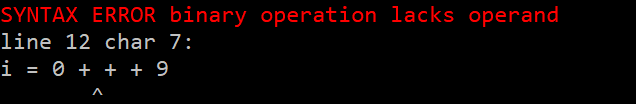
righthalf = alist( mid: )

mergeSort(lefthalf)

mergeSort(righthalf)

i = 0 + + +9

*Листинг. 4.3. Лишний оператор*



*Рис. 4.3. Ошибка отловлена*

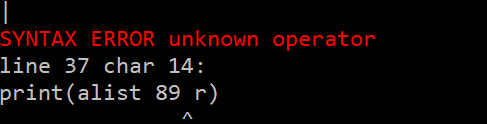
4) Введем аргументы в функцию, не разделяя их запятыми:

…

mergeSort(alist)

print(alist 89 r)

*Листинг. 4.4. Пробуем ввести в случайное место число*



*Рис. 4.4. Ошибка отловлена*

5) Введем лишний оператор «\*»:

while i < len(lefthalf) and j < len(righthalf) :

if lefthalf(i) < righthalf(j) :

alist(k) = lefthalf(i)

i = i + 1

else :

alist(k+ \*m) = righthalf(j)

j = j + 1

k = k + 1

*Листинг. 4.5. Лишний оператор*



*Рис. 4.5. Ошибка отловлена*

## 

## Выводы

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.

Язык разбора Python — [высокоуровневый язык программирования](https://ru.wikipedia.org/wiki/Высокоуровневый_язык_программирования) общего назначения, ориентированный на повышение производительности разработчика и читаемости кода, в то же время [стандартная библиотека](https://ru.wikipedia.org/wiki/Стандартная_библиотека_Python) включает большой объём полезных функций. Основные архитектурные черты — [динамическая типизация](https://ru.wikipedia.org/wiki/Динамическая_типизация), [автоматическое управление памятью](https://ru.wikipedia.org/wiki/Сборка_мусора_(программирование)), полная [интроспекция](https://ru.wikipedia.org/wiki/Интроспекция_(программирование)), механизм [обработки исключений](https://ru.wikipedia.org/wiki/Обработка_исключений), поддержка [многопоточных вычислений](https://ru.wikipedia.org/wiki/Многопоточность) и удобные высокоуровневые [структуры данных](https://ru.wikipedia.org/wiki/Структура_данных).

Код в Python организовывается в функции и [классы](https://ru.wikipedia.org/wiki/Класс_(программирование)), которые могут объединяться в [модули](https://ru.wikipedia.org/wiki/Модуль_(программирование)) (они в свою очередь могут быть объединены в пакеты).

## Приложение. Код программы

namespace ConsoleApp1

{

class SyntaxAnalizer

{

protected int OpenedBracketsLevel = 0;

protected int CurrentBlockLevel = 0;

public ExpressionNode Analyse(IEnumerable<Token> tokens, out bool startNewBlock, out bool isElifElseNode)

{

OpenedBracketsLevel = 0;

startNewBlock = false;

isElifElseNode = false;

var firstToken = tokens.FirstOrDefault();

if (firstToken?.IsBlockOpeningOperation == true)

{

startNewBlock = true;

isElifElseNode = firstToken.TokenType == TokenTypes.ELSE || firstToken.TokenType == TokenTypes.ELIF;

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

{

var t = tokens.LastOrDefault();

throw new SyntaxErrorException("colon expected", t.Value, t.CodeLineIndex, t.CodeLineNumber);

}

}

ExpressionNode root = BuildTree(tokens);

if (OpenedBracketsLevel != 0)

{

throw new SyntaxErrorException("brackets do not match", tokens.Last().Value, tokens.Last().CodeLineIndex, tokens.Last().CodeLineNumber);

}

return root;

}

protected ExpressionNode BuildTree(IEnumerable<Token> tokens, ExpressionNode parent = null)

{

ExpressionNode root = null;

ExpressionNode left = null;

Token token = tokens.FirstOrDefault();

if (token is null)

return null;

if (token.IsConstant || token.TokenType == Token.TokenTypes.ID || token.TokenType == Token.TokenTypes.BUILT\_IN\_FUNCTION)

{

left = new ExpressionNode()

{

Operator = token,

Type = ExpressionNode.TokensToExpressionTypes.GetOrDefault(token.TokenType, ExpressionNode.ExpressionTypes.UNKNOWN)

};

var tt = tokens.ElementAtOrDefault(1)?.TokenType;

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

{

root = left;

left = null;

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

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

}

else if (tt == Token.TokenTypes.COLON)

{

root = left;

left = null;

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

// TODO: maybe throw error here if there is something after COLON

}

else

{

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

left.Parent = root;

}

}

else if (token.IsOpeningBracket)

{

this.OpenedBracketsLevel++;

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

root.OperatorPriority++;

}

else if (token.IsClosingBracket)

{

this.OpenedBracketsLevel--;

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

if (root != null)

root.OperatorPriority--;

}

else if (token.IsOperation)

{

root = new ExpressionNode()

{

Operator = token,

Type = ExpressionNode.TokensToExpressionTypes.GetOrDefault(token.TokenType, ExpressionNode.ExpressionTypes.UNKNOWN)

};

if (token.TokenType == Token.TokenTypes.MULTIPLICATION || token.TokenType == Token.TokenTypes.DIVISION)

{

root.OperatorPriority++;

}

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

}

if (root is null)

{

if (left is null)

return null;

left.Parent = parent;

return left;

}

root.Parent = parent;

if (left != null)

root.InsertDeepLeft(left);

if (root.Right != null && root.Operator.IsOperation && root.Right.Operator.IsOperation && root.OperatorPriority > root.Right.OperatorPriority)

return root.LeftRotation();

return root;

}

public static ExpressionNode ValidateNode(ExpressionNode node)

{

switch (node.Type)

{

case ExpressionNode.ExpressionTypes.BINARY\_OPERATION:

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

{

throw new SyntaxErrorException(

"binary operation lacks operand",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

break;

case ExpressionNode.ExpressionTypes.BLOCK\_OPENING\_CONDITIONAL\_OPERATION:

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

throw new SyntaxErrorException(

"conditional operator wrong usage",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

break;

case ExpressionNode.ExpressionTypes.UNKNOWN:

throw new SyntaxErrorException(

"unknown expression",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

case ExpressionNode.ExpressionTypes.OPERAND:

if (node.Left != null)

throw new SyntaxErrorException(

"unknown operator",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

break;

default:

break;

}

return node;

}

public class SyntaxErrorException : FormatException

{

public string Value { get; set; }

public int PositionInLine { get; set; }

public int LineNumber { get; set; }

public SyntaxErrorException(string message, string value, int positionInLine, int lineNumber) : base(message)

{

Value = value;

PositionInLine = positionInLine;

LineNumber = lineNumber;

}

}

public class ExpressionNode

{

public ExpressionNode Left = null;

public Token Operator = null;

public ExpressionTypes Type;

public int OperatorPriority = 0;

public ExpressionNode Right = null;

public ExpressionNode Parent = null;

public TreeList<ExpressionNode> Block = new TreeList<ExpressionNode>(null);

public ExpressionNode LeftRotation()

{

ExpressionNode newRoot = new ExpressionNode()

{

Right = this.Right.Right,

Operator = this.Right.Operator,

Type = this.Right.Type,

Parent = this.Parent

};

newRoot.Left = new ExpressionNode()

{

Left = this.Left,

Right = this.Right.Left,

Operator = this.Operator,

Type = this.Type,

Parent = newRoot

};

return newRoot;

}

public void InsertDeepLeft(ExpressionNode node)

{

ExpressionNode temp = this;

while (!(temp.Left is null))

{

temp = temp.Left;

}

temp.Left = node;

}

public override string ToString()

{

return $"({Operator.ToString()})";

}

public enum ExpressionTypes

{

UNKNOWN,

UNARY\_OPERATION,

BINARY\_OPERATION,

BLOCK\_OPENING\_CONDITIONAL\_OPERATION,

BLOCK\_OPENING\_OPERATION,

FUNCTION\_CALL,

FUNCTION\_DEF,

OPERAND

};

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

{

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[TokenTypes.ID] = ExpressionTypes.OPERAND,

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

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

};

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

using static ConsoleApp1.Token;

using static ConsoleApp1.LexicalAnalizer;

using ConsoleTables;

namespace ConsoleApp1

{

class Program

{

static void PrintTokensDictionary(Dictionary<string, Token> dictionary)

{

ConsoleTable consoleTable = new ConsoleTable("TOKEN", "DESCRIPTION");

foreach (Token token in dictionary.Values)

{

consoleTable.AddRow(token.Value, token.DescriptionString);

}

consoleTable.Write();

}

static string PrintNodeWithChildren(SyntaxAnalizer.ExpressionNode node, string indentation)

{

if (node == null)

{

return "";

}

SyntaxAnalizer.ValidateNode(node);

StringBuilder stringBuilder = new StringBuilder();

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

if (node.Left != null)

stringBuilder.Append(PrintNodeWithChildren(node.Left, indentation + "\\"));

if (node.Right != null)

stringBuilder.Append(PrintNodeWithChildren(node.Right, indentation + "\\"));

return stringBuilder.ToString();

}

static void PrintSyntaxTree(IEnumerable<SyntaxAnalizer.ExpressionNode> nodes, int nestingLevel = 1)

{

string indentation = new String('|', nestingLevel);

foreach (var node in nodes)

{

Console.Write(PrintNodeWithChildren(node, indentation));

Console.WriteLine(indentation);

PrintSyntaxTree(node.Block, nestingLevel+1);

}

}

static string errorDescription(int indexInCodeLine, string codeLine)

{

StringBuilder stringBuilder = new StringBuilder(codeLine);

stringBuilder.AppendLine();

stringBuilder.Append(new string(' ', indexInCodeLine));

stringBuilder.Append('^');

return stringBuilder.ToString();

}

static void DoTheJob(IEnumerable<string> codeLines)

{

Dictionary<string, Token> constants = new Dictionary<string, Token>();

Dictionary<string, Token> variables = new Dictionary<string, Token>();

Dictionary<string, Token> operators = new Dictionary<string, Token>();

Dictionary<string, Token> keywords = new Dictionary<string, Token>();

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

// running lexical analysis

TreeList<SyntaxAnalizer.ExpressionNode> tree = new TreeList<SyntaxAnalizer.ExpressionNode>(null);

TreeList<SyntaxAnalizer.ExpressionNode> currentBlock = tree;

int lineNumber = 0;

SyntaxAnalizer sa = new SyntaxAnalizer();

LexicalAnalizer la = new LexicalAnalizer();

int previousLineIndentation = 0;

foreach (string line in codeLines)

{

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

if (construction.Tokens.Count == 0)

{

lineNumber++;

continue;

}

for (int i = 0; i < construction.Tokens.Count; i++)

{

Token token = construction.Tokens[i];

if (token.IsReservedIdToken)

keywords.TryAdd(token.Value, token);

else if (token.IsOperation)

operators.TryAdd(token.Value, token);

else if (token.IsConstant)

constants.TryAdd(token.Value, token);

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

{

variables.TryAdd(token.Value, token);

}

}

if (construction.HasErrors)

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("\t\t ERRORS");

Console.ResetColor();

foreach (LexicalError error in construction.Errors)

{

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

Console.WriteLine(error.Description);

}

Console.Read();

Environment.Exit(1);

}

SyntaxAnalizer.ExpressionNode node = null;

bool isElifElseNode = false;

bool newBlockToOpen = false;

node = sa.Analyse(construction.Tokens, out newBlockToOpen, out isElifElseNode);

int indentationDiff = previousLineIndentation - construction.Indentation;

if (indentationDiff > 0)

{

for (int i = previousLineIndentation-1; i >= construction.Indentation; i--)

{

currentBlock = currentBlock.Parent;

if (currentBlock.Indentation == i)

break;

}

// currentBlock = currentBlock.Parent; // TODO: create parent relationship between BLOCKS to support >1 level nesting

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

{

throw new SyntaxAnalizer.SyntaxErrorException(

"elif block not allowed here",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

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

{

throw new SyntaxAnalizer.SyntaxErrorException(

"else block not allowed here",

line,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

}

previousLineIndentation = construction.Indentation;

lineNumber++;

if (newBlockToOpen)

{

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

{

throw new SyntaxAnalizer.SyntaxErrorException(

"lacks IF clause for elif|else block to appear",

node.Operator.Value,

node.Operator.CodeLineIndex,

node.Operator.CodeLineNumber

);

}

currentBlock.Add(node);

currentBlock.Last().Block = new TreeList<SyntaxAnalizer.ExpressionNode>(currentBlock);

currentBlock = currentBlock.Last().Block;

currentBlock.Indentation = construction.Indentation;

continue;

}

currentBlock.Add(node);

}

if (errors.Any())

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine("\t\t ERRORS");

Console.ResetColor();

foreach (LexicalError error in errors)

{

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

Console.WriteLine(error.Description);

}

}

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

PrintSyntaxTree(tree);

// console tables output block

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

PrintTokensDictionary(constants);

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

PrintTokensDictionary(variables);

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

PrintTokensDictionary(keywords);

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

PrintTokensDictionary(operators);

}

static void Main(string[] args)

{

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

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

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

try

{

DoTheJob(codeLines);

}

catch (SyntaxAnalizer.SyntaxErrorException e)

{

Console.ForegroundColor = ConsoleColor.Red;

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

Console.ResetColor();

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

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

}

catch (InvalidOperationException e)

{

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

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

}

Console.Read();

}

}

}