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. 753504

Осипик И.Ф.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2020

**Содержание**

[1. Цель Работы 3](#_Toc8656852)

[2. Общие Теоретические Сведения 5](#_Toc8656853)

[2.1 Основные функции семантического анализатора 5](#_Toc8656854)

[2.2 Идентификация 6](#_Toc8656855)

[2.3 Контроль и эквивалентность типов. 7](#_Toc8656856)

[3. Результат Работы 9](#_Toc8656857)

[4. Вывод 11](#_Toc8656858)

[Приложение. Код программы 12](#_Toc8656859)

# 

# Цель Работы

Синтаксический анализ — процесс разбора структуры на предмет ошибок, позволенных синтаксисом, но не семантикой языка программирования.

Цель работы — используя наработки предыдущей работы расширить классы, составляющие синтаксическое дерево, поведением и сделать возможным интерпретацию программы на выбранном языке.

Требования: в случае семантической корректности программы выдать соответствующее сообщение, иначе выдать сообщение с информацией об ошибке и остановить работу. В отличие от предыдущих работ в случае возникновения семантической ошибки продолжить выполнение невозможно — нет необходимой информации и теряется целостность данных, что порождает множество дальнейших ошибок.

def mergeSort(alist) :

print("Splitting ",alist)

if len(alist) > 1:

mid = int(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)

alist = [54, 26, 93, 17]

mergeSort(alist)

print(alist)

*Листинг 1.1. Исходный код на Python.*

# Общие Теоретические Сведения

### 2.1 Основные функции семантического анализатора

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*. Таблица формируется на этапе лексического анализа, где в нее помещаются все уникальные имена, распознанные сканером. Во время семантического анализа для каждого имени заносятся все данные, полученные из текста программы (тип идентификатора, тип значений и т.д.).

2) *Выделение неявно заданной информации*. В представлении программ некоторые данные об элементах программы не указаны явно.

3*) Обнаружение ошибок.* Синтаксический анализ определяет корректность отдельных конструкций и программы в целом с точки зрения формальных правил используемого языка, но и здесь могут быть ошибки (не согласованы типы правой и левой частей оператора присваивания, несколько одинаковых меток и т.д.).

4) *Выполнение некоторых операций программы*. Присваивание начальных значений; Действия с константами; Обработка директив компилятора.

5) *Формирование внутренней формы программы*. Часто используются такие формы, как семантическое дерево, польская запись, список тетрад.

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). Например, семантическая программа, обрабатывающая описания переменных и массивов, вносит их типы в таблицу имен, далее эта информация используется при работе семантических программ, анализирующих выражения и операторы присваивания.

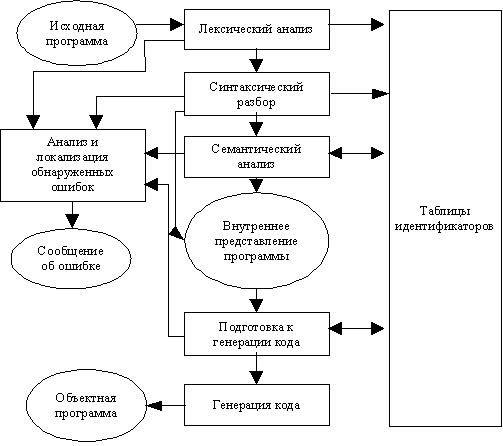


Рисунок 2.1.1. Схема работы интерпретатора

### 2.2 Идентификация

Идентификация идентификаторов – одна из задач, решение которой необходимо для проверки правильности использования типов.

Понятно, что невозможно убедиться в правильности использования типов в какой-нибудь конструкции до тех пор, пока не определим типы всех ее составных частей. Например, для того, чтобы выяснить правильность оператора присваивания, мы должны знать типы его получателя (левой части) и источника (правой части). Для того, чтобы выяснить, каков тип идентификатора, являющегося, например, получателем присваивания, надо понять, каким образом этот идентификатор был объявлен в программе.

Каждое вхождение идентификатора в программу является либо определяющим, либо использующим. Под определяющим вхождением идентификатора понимается его вхождение в описание, например, int i.   
Все остальные вхождения являются использующими, например, i = 5 или i+13.

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.

Это можно сделать несколькими путями:

• создать узел в синтаксическом дереве для конструкции «описание идентификатора» и запоминать информацию о типе идентификатора в этом узле;

• создать таблицу идентификаторов (IdTab) и в ней запоминать информацию о тип идентификатора.

Почему нам может потребоваться новая таблица? Понятно, что если транслируемая программа имеет блочную структуру, и/или язык допускает создание и использование перегруженных идентификаторов (overloading), то в таблице представлений (таблица представлений сопоставляет некоторому используемому в компиляторе обозначению идентификатора его представление в программе) информацию о типе идентификатора хранить нельзя, поскольку в этой таблице каждая лексема встречается только один раз. Таким образом, нам потребуется новая таблица для хранения информации об определяющих вхождениях идентификаторов.

### 2.3 Контроль и эквивалентность типов.

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.

Понятно, что динамический контроль типов приводит к увеличению размера и времени исполнения объектной программы, и уменьшению ее надежности. Язык программирования называется языком со статическим контролем типов или строго типизированным языком (strongly typed language), если тип любого выражения может быть определен во время трансляции, т. е. если можно гарантировать, что объектная программа выполняется без типовых ошибок. К числу строго типизированных языков относится, например, Паскаль.

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

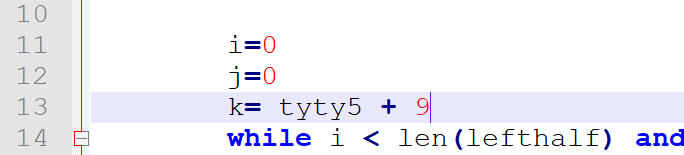
Структурная эквивалентность типов (Structural equivalence of types). Два типа называются эквивалентными если они являются одинаковыми примитивными типами, либо они были сконструированы с применением одного и того же конструктора к структурно эквивалентным типам. Иными словами, два типа структурно эквивалентны тогда и только тогда, когда они идентичны.

Второй подход связан с понятием эквивалентности имен (name equivalence). В этом случае каждое имя типа рассматривается как уникальный тип, таким образом, два имени типов эквивалентны, если они идентичны.

# 

# Результат работы

1. Ошибка – неизвестный идентификатор.



Листинг 3.1 – Код с ошибкой «Неизвестный идентификатор»

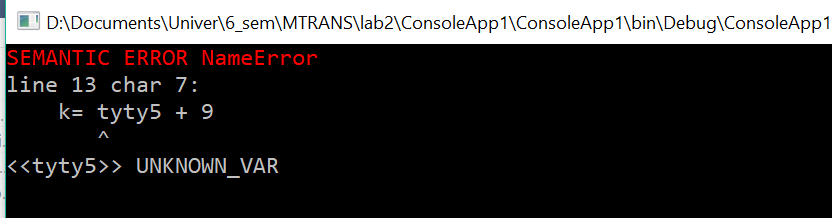
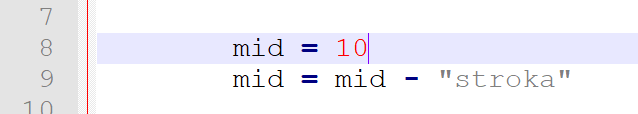


Рис. 3.1 – Ошибка отловлена

1. Ошибка – разность разных типов.



Листинг 3.2 – Код с делением на 0

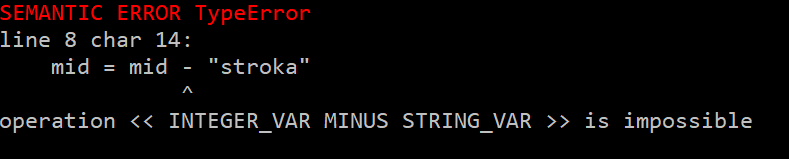
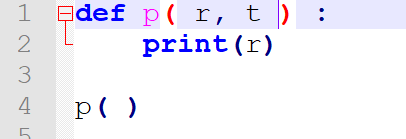


Рис 3.2 – Ошибка отловлена

1. Ошибка – вызов функции с отсутствием аргументов.



Листинг 3.3 Функция и ее вызов.

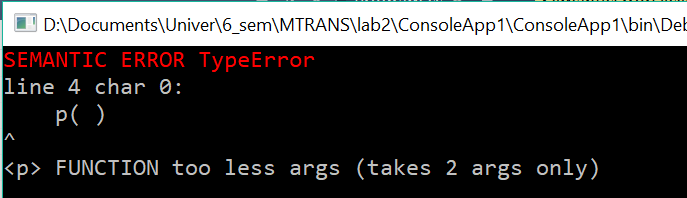


Рис 3.3 – Ошибка отловлена

# 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.

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.

# 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));

}

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.Threading.Tasks;

namespace ConsoleApp1

{

class SemanticItem

{

public string Name { get; set; }

public VarTypes VarType { get; set; }

public FunctionSpecification FSpecification { 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, 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

};

public struct FunctionSpecification

{

public VarTypes ReturnType { get; set; }

public int MaxArgumentsAmount { get; set; }

public int MinArgumentsAmount { get; set; }

}

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 }

};

}

}

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();

}

}

}