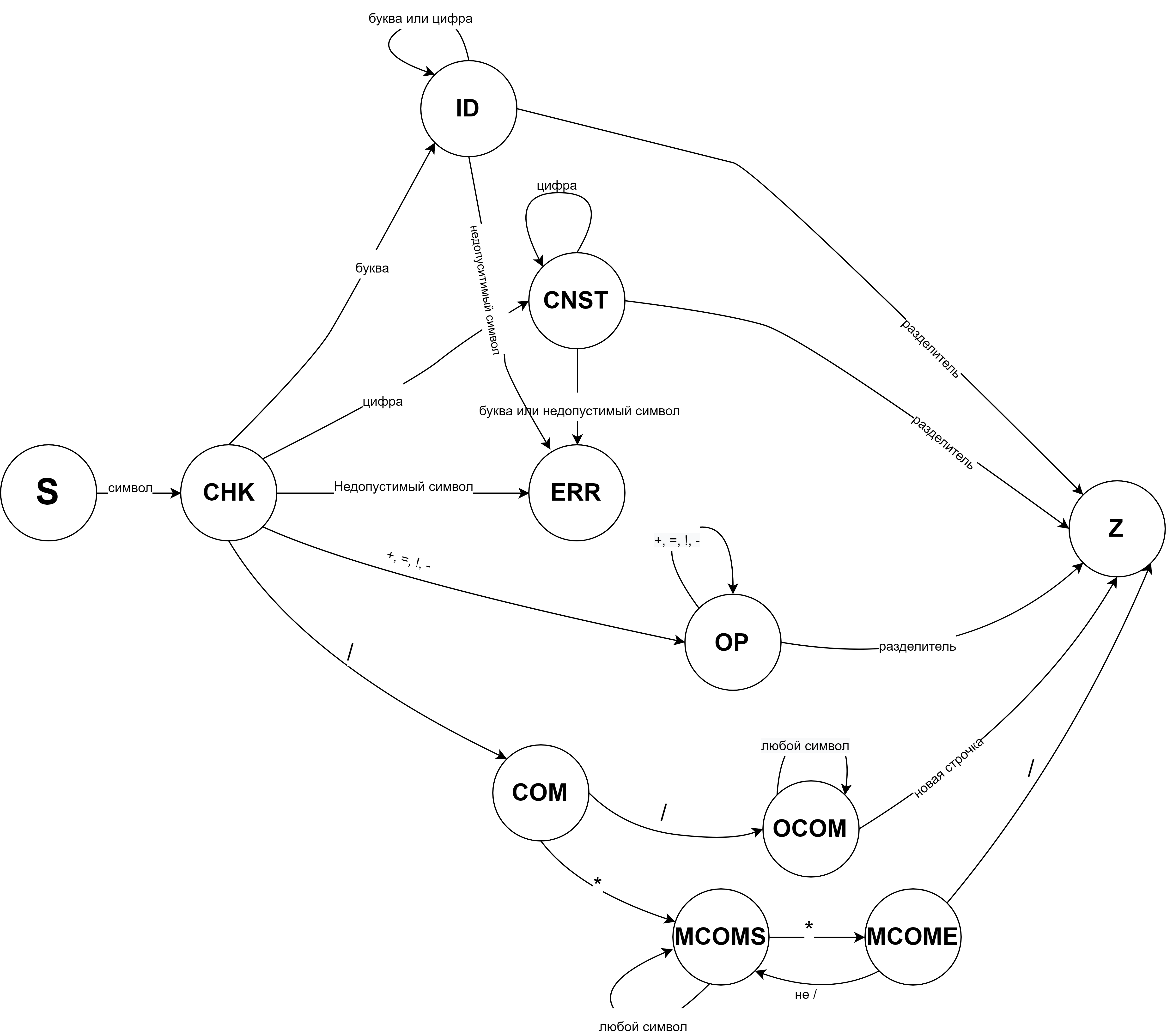
1. **Цель работы**

Изучить методы лексического анализа. Получить представление о методах обработки лексических ошибок. Научиться проектировать сканер на основе детерминированных конечных автоматов.

1. **Входные и выходные данные**

На вход подается имя файла, в котором содержится исходный код программы на языке С. Результатом работы программы являются два файла - файл токенов и файл ошибок. Файл токена представлен в виде (тип токена, (позиция в таблице, позиция в строке)). Тип токена - целочисленное значение (10 - для ключевого слова, 20 для разделителя, 30 для идентификатора, 40 для константы, 50 - для операции)

1. **Детерминированный конечный автомат**

**Описание состояний:**

**S -** начальное состояние;

**СHK -** проверка типа символа;

**ERR -** ошибка;

**СNST -** цепочка константа;

**ID -** цепочка идентификатор или ключевое слово;

**COM -** проверка на комментарий;

**OCOM -** однострочный комментарий;

**MCOMS -** начало многострочного комментария;

**MCOME -** конец многострочного комментария;

1. **Алгоритм**

Для каждой строки

Для каждого символа из строки

Если открыт комментарий «//» - перейти к следущей строке

Если открыт многострочный комментарий

Если текущий символ == «\*»

Добавить символ к текущему слову

Перейти к следующему символу

Если текущий символ == «/» и текущее слово == «\*»

закрыть многострочный комментарий

Текущее слово = «»

Если символ = начало слова

тип текущего слова = тип текущего символа

следущий символ - не начало слова

если тип текущего символа = разделитель

следущий символ - начало слова

если текущее слово не пустая строка

добавить текущее слово в файл токенов

текущее слово = «»

добавить разделитель в файл токенов

если тип текущего символа = константа

если тип текущего слова = операция

добавить текущее слово в файл токенов

текущее слово = текущий символ

тип текущего слова = тип текущего символа

Перейти к следующему символу

добавить к текущему слову текущий символ

если тип текущего символа = ключевое слово или идент.

если тип текущего слова = константа

Ошибка: недопустимые символы в числе

Перейти к следующему символу

если тип текущего слова != тип текущего символа

добавить текущее слово в файл токенов

текущее слово = текущий символ

тип текущего слова = тип текущего символа

Перейти к следующему символу

добавить к текущему слову текущий символ

если тип текущего символа = операция

если тип текущего слова != тип текущего символа

добавить текущее слово в файл токенов

текущее слово = текущий символ

тип текущего слова = тип текущего символа

Перейти к следующему символу

если текущее слово == «/»

если текущий символ == «/»

открыть однострочный комментарий

текущее слово == «»

перейти к следующему символу

если текущий символ == «\*»

открыть многострочный комментарий

текущее слово == «»

перейти к следующему символу

| Таблица ключевых слов | |
| --- | --- |
| 0 | do |
| 1 | int |
| 2 | main |
| 3 | void |
| 4 | while |

| Таблица операций | |
| --- | --- |
| 0 | - |
| 1 | != |
| 2 | + |
| 3 | += |
| 4 | = |
| 5 | -= |
| 6 | == |

| Таблица разделителей | |
| --- | --- |
| 0 |  |
| 1 | ( |
| 2 | ) |
| 3 | ; |
| 4 | { |
| 5 | } |

добавить к текущему слову текущий символ

| **Таблица идентификаторов для 1 теста** | |
| --- | --- |
| 97 | a |

| **Таблица констант для 1 теста** | |
| --- | --- |
| 0 | 10 |

| **Таблица идентификаторов для 2, 3, 4 теста** | |
| --- | --- |
| 97 | a |
| 98 | b |
| 99 | c |

| **Таблица констант для 2, 3, 4 теста** | |
| --- | --- |
| 0 | 10 |
| 1 | 11 |

если тип текущего слова = неправильный символ

Ошибка: недопустимые символы

если открыт многострочный комментарий

Ошибка: незакрытый комментарий

1. **Тесты**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Текст исходной программы** | **Содержимое файла токенов** | **Содержимое файла ошибок** | **Комментарий** |
| **1** | int main() {  int a=10; // v = 10  /\* int b = 20;  int c=a+b; \*/  } | (10, (1, -1))  (10, (2, -1))  (20, (1, -1))  (20, (2, -1))  (20, (4, -1))  (10, (1, -1))  (30, (97, 0))  (50, (4, -1))  (40, (0, -1))  (20, (3, -1))  (20, (5, -1)) |  | Тест с лексически верным исходным кодом |
| **2** | int main() {  int a = 10; /\*  int b = 20;  int c=a+b;  } | (10, (1, -1))  (10, (2, -1))  (20, (1, -1))  (20, (2, -1))  (20, (4, -1))  (10, (1, -1))  (30, (97, 0))  (50, (4, -1))  (40, (0, -1))  (20, (3, -1))  (20, (5, -1)) | Tokenizer reaches end with not closed comment | Тест на определение незакрытого многострочного комментария |
| **3** | int main() {  int a = 10; #  int b = 20;  #  int c=a+b;  } | (10, (1, -1))  (10, (2, -1))  (20, (1, -1))  (20, (2, -1))  (20, (4, -1))  (10, (1, -1))  (30, (97, 0))  (50, (4, -1))  (40, (0, -1))  (20, (3, -1))  (10, (1, -1))  (30, (98, 0))  (50, (4, -1))  (40, (1, -1))  (20, (3, -1))  (10, (1, -1))  (30, (99, 0))  (50, (4, -1))  (30, (97, 0))  (50, (2, -1))  (30, (98, 0))  (20, (3, -1))  (20, (5, -1)) | Forribden character at line 1, column 11  Forribden character at line 3, column 0 | Тест на определение недопустимых символов |
| **4** | int main() {  int a = 1$0;  int b = 20a;  int c=a+b;  } | (10, (1, -1))  (10, (2, -1))  (20, (1, -1))  (20, (2, -1))  (20, (4, -1))  (10, (1, -1))  (30, (97, 0))  (50, (4, -1))  (40, (0, -1))  (20, (3, -1))  (10, (1, -1))  (30, (98, 0))  (50, (4, -1))  (40, (1, -1))  (20, (3, -1))  (10, (1, -1))  (30, (99, 0))  (50, (4, -1))  (30, (97, 0))  (50, (2, -1))  (30, (98, 0))  (20, (3, -1))  (20, (5, -1)) | Forribden character at line 1, column 8  Using non-allowed characters in number at line 2, column 12 | Тест на недопустимые символы в числе и идентификаторах |

1. **Текст программы**

***Файл program.cs***

using System;

namespace Assembler\_Translator

{

class Program

{

static void Main(string[] args)

{

var translator = new Translator();

translator.tokenize("testProgram1.txt");

}

}

}

***Файл translator.cs***

using System.IO;

using System.Collections.Generic;

using System;

namespace Assembler\_Translator

{

class Translator

{

private DynamicTable dynamicTable = new DynamicTable();

private StaticTable staticTable = new StaticTable();

enum TypeOfChar

{

separator, keywordOrIdentificator, constant, operations, wrongCharacter

}

public List<List<Token>> tokenize(string fileName)

{

var errorsList = new List<TokenizeError>();

var finalList = new List<List<Token>>();

string[] programLines = File.ReadAllLines(fileName);

bool isOpenedMultilineComment = false;

int lineIndex = 0;

foreach (var programLine in programLines)

{

int columnIndex = -1;

List<Token> lineTokens = new List<Token>();

if (programLine.Length != 0)

{

bool isStartOfWord = true;

bool isOpenedComment = false;

TypeOfChar typeOfCurrentWord = getTypeOfChar(programLine[0]);

string currentWord = "";

foreach (char symbol in programLine)

{

columnIndex++;

if (isOpenedComment) break; // Если обнаружено начало комментария "//" - то прекратить считывание текущей строки, тк в этом нет смысла

if (isOpenedMultilineComment)

{

if (symbol == '\*')

{

currentWord = symbol.ToString();

continue;

}

if (symbol == '/' && currentWord.Length != 0)

{

isOpenedMultilineComment = false;

currentWord = "";

}

currentWord = "";

continue;

}

if (isStartOfWord)

{

typeOfCurrentWord = getTypeOfChar(symbol);

isStartOfWord = false;

}

var typeOfCurrentChar = getTypeOfChar(symbol);

switch (typeOfCurrentChar)

{

case TypeOfChar.separator:

isStartOfWord = true;

if (currentWord.Length == 0)

{

var separator = addWord(TypeOfChar.separator, symbol.ToString());

if (separator.HasValue) lineTokens.Add(separator.Value);

continue;

}

lineTokens.Add(addWord(typeOfCurrentWord, currentWord).Value);

currentWord = "";

var separatorToken = addWord(TypeOfChar.separator, symbol.ToString());

if (separatorToken.HasValue) lineTokens.Add(separatorToken.Value);

break;

case TypeOfChar.constant:

if (typeOfCurrentWord == TypeOfChar.operations)

{

lineTokens.Add(addWord(typeOfCurrentWord, currentWord).Value);

currentWord = "";

currentWord += symbol;

typeOfCurrentWord = TypeOfChar.constant;

continue;

}

currentWord += symbol;

continue;

case TypeOfChar.keywordOrIdentificator:

if (typeOfCurrentWord == TypeOfChar.constant)

{

errorsList.Add(new TokenizeError(TokenizeErrorType.wrongFormatOfNumber, lineIndex, columnIndex));

continue;

}

if (typeOfCurrentWord != typeOfCurrentChar)

{

lineTokens.Add(addWord(typeOfCurrentWord, currentWord).Value);

currentWord = "";

currentWord += symbol;

typeOfCurrentWord = typeOfCurrentChar;

continue;

}

currentWord += symbol;

break;

case TypeOfChar.operations:

if (typeOfCurrentWord != typeOfCurrentChar)

{

lineTokens.Add(addWord(typeOfCurrentWord, currentWord).Value);

currentWord = "";

currentWord += symbol;

typeOfCurrentWord = typeOfCurrentChar;

continue;

}

if (currentWord == "/")

{

if (symbol == '/')

{

isOpenedComment = true;

currentWord = "";

break;

}

if (symbol == '\*')

{

isOpenedMultilineComment = true;

currentWord = "";

break;

}

}

currentWord += symbol;

break;

case TypeOfChar.wrongCharacter:

errorsList.Add(new TokenizeError(TokenizeErrorType.forbiddenСharacter, lineIndex, columnIndex));

break;

}

}

}

finalList.Add(lineTokens);

lineIndex++;

}

if (isOpenedMultilineComment)

errorsList.Add(new TokenizeError(TokenizeErrorType.notClosedComment, null, null));

#if DEBUG

using (StreamWriter file = new StreamWriter("tokens.txt"))

foreach (var tokenLine in finalList)

file.WriteLine(String.Join(" ", tokenLine.ToArray()));

using (StreamWriter file = new StreamWriter("errors.txt"))

foreach (var error in errorsList)

file.WriteLine($"{error.localizedError()}");

foreach (var item in staticTable.keywords)

Console.WriteLine(item);

foreach (var item in staticTable.operations)

Console.WriteLine(item);

foreach (var item in staticTable.separators)

Console.WriteLine(item);

#endif

return finalList;

}

private Token? addWord(TypeOfChar typeOfWord, string word)

{

switch (typeOfWord)

{

case TypeOfChar.constant:

var intValue = int.Parse(word);

return dynamicTable.addConstant(intValue);

case TypeOfChar.keywordOrIdentificator:

var keywordToken = staticTable.getIndexOf(word);

if (keywordToken.HasValue) return keywordToken;

return dynamicTable.addIdentificator(word);

case TypeOfChar.operations:

return staticTable.getIndexOf(word);

case TypeOfChar.separator:

if (word == " " || word == "\t") return null;

return staticTable.getIndexOf(word);

default:

return null;

}

}

private TypeOfChar getTypeOfChar(char symbol)

{

if (staticTable.separators.Contains(symbol.ToString()) || symbol == '\t') return TypeOfChar.separator;

if (staticTable.allowedAlphabeth.Contains(symbol)) return TypeOfChar.keywordOrIdentificator;

if (staticTable.allowedNumbers.Contains(symbol)) return TypeOfChar.constant;

if (staticTable.operationsChars.Contains(symbol.ToString())) return TypeOfChar.operations;

return TypeOfChar.wrongCharacter;

}

}

}

***Файл token.cs***

using System;

namespace Assembler\_Translator

{

enum TokenType: int

{

keyword = 10,

separator = 20,

identificator = 30,

constant = 40,

operation = 50

}

public enum TokenizeErrorType

{

forbiddenСharacter,

wrongFormatOfNumber,

notClosedComment

}

struct TokenPosition

{

int positionInTable;

int positionInRow;

public TokenPosition(int positionInTable, int positionInRow)

{

this.positionInTable = positionInTable;

this.positionInRow = positionInRow;

}

public override string ToString() => $"({positionInTable}, {positionInRow})";

}

struct Token

{

TokenType tokenType;

TokenPosition tokenPosition;

public Token(TokenType tokenType, TokenPosition tokenPosition)

{

this.tokenType = tokenType;

this.tokenPosition = tokenPosition;

}

public Token(TokenType tokenType, int positionInTable, int positionInRow)

{

this.tokenType = tokenType;

this.tokenPosition = new TokenPosition(positionInTable, positionInRow);

}

public override string ToString() => $"({(int)tokenType}, {tokenPosition})";

}

public struct TokenizeError

{

public TokenizeErrorType error;

public int? line;

public int? column;

public TokenizeError(TokenizeErrorType error, int? line, int? column)

{

this.error = error;

this.line = line;

this.column = column;

}

}

public static class TokenizeErrorExtantions

{

public static string localizedError(this TokenizeError error)

{

return $"{error.error.localizedError(error.line, error.column)}";

}

}

public static class TokenizeErrorTypeExtensions

{

public static string localizedError(this TokenizeErrorType error, int? line, int? column)

{

switch (error)

{

case TokenizeErrorType.forbiddenСharacter:

return $"Forribden character at line {line}, column {column}";

case TokenizeErrorType.notClosedComment:

return $"Tokenizer reaches end with not closed comment";

case TokenizeErrorType.wrongFormatOfNumber:

return $"Using non-allowed characters in number at line {line}, column {column}";

default:

return "";

}

}

}

}

***Файл stringExtentions.cs***

using System;

namespace Assembler\_Translator

{

public static class StringExtentions

{

public static int CalculateHash(this string str, int upperBound)

{

int total = 0;

char[] charArray;

charArray = str.ToCharArray();

foreach (char singleChar in charArray)

total += 11 \* total + (int)singleChar;

total %= upperBound;

return total;

}

}

}

***Файл staticTable.cs***

using System;

using System.IO;

using System.Collections.Generic;

namespace Assembler\_Translator

{

class StaticTable

{

private const string keywordsTableFileName = "keywords.txt";

private const string separatorsTableFileName = "separators.txt";

private const string operationsTableFileName = "operations.txt";

private const string operationsCharsTableFileName = "operationsChars.txt";

private const string allowedAlphabethCharsTableFileName = "allowedAlphabeth.txt";

private const string allowedNumbersCharsTableFileName = "allowedNumbers.txt";

public List<String> keywords { get; private set; }

public List<String> separators { get; private set; }

public List<String> operations { get; private set; }

public List<String> operationsChars { get; private set; }

public HashSet<Char> allowedAlphabeth { get; private set; }

public HashSet<Char> allowedNumbers { get; private set; }

public StaticTable()

{

keywords = new List<string>();

separators = new List<string>();

operations = new List<string>();

operationsChars = new List<string>();

allowedAlphabeth = new HashSet<Char>();

allowedNumbers = new HashSet<Char>();

string[] keywordLines = File.ReadAllLines(keywordsTableFileName);

string[] separatorsLines = File.ReadAllLines(separatorsTableFileName);

string[] operationsLines = File.ReadAllLines(operationsTableFileName);

string[] operationsCharsLines = File.ReadAllLines(operationsCharsTableFileName);

string[] allowedAlphabethLines = File.ReadAllLines(allowedAlphabethCharsTableFileName);

string[] allowedNumbersLines = File.ReadAllLines(allowedNumbersCharsTableFileName);

foreach (string keyword in keywordLines)

{

if (keywords.Contains(keyword)) continue;

keywords.Add(keyword);

}

foreach (string separator in separatorsLines)

{

if (separators.Contains(separator)) continue;

separators.Add(separator);

}

foreach (var operation in operationsLines)

{

if (operations.Contains(operation)) continue;

operations.Add(operation);

}

foreach (var operationsChar in operationsCharsLines)

{

if (operationsChars.Contains(operationsChar)) continue;

operationsChars.Add(operationsChar);

}

foreach (string line in allowedAlphabethLines)

foreach (var symbol in line)

allowedAlphabeth.Add(symbol);

foreach (string line in allowedNumbersLines)

foreach (var symbol in line)

allowedNumbers.Add(symbol);

keywords.Sort();

separators.Sort();

operations.Sort();

}

public Token? getIndexOf(string word)

{

int keywordIndex = keywords.FindIndex(keyword => keyword == word);

if (keywordIndex != -1) return new Token(TokenType.keyword, keywordIndex, -1);

int separatorIndex = separators.FindIndex(keyword => keyword == word);

if (separatorIndex != -1) return new Token(TokenType.separator, separatorIndex, -1);

int operationIndex = operations.FindIndex(operation => operation == word);

if (operationIndex != -1) return new Token(TokenType.operation, operationIndex, -1);

return null;

}

public bool isOperation(char symbol) => operationsChars.Contains(symbol.ToString());

}

}

***Файл dynamicTable.cs***

using System;

using System.Collections.Generic;

namespace Assembler\_Translator

{

class DynamicTable

{

private List<String>[] identificators;

private List<int> constants;

private const int dynamicTableSize = 100;

public DynamicTable()

{

constants = new List<int>();

identificators = new List<string>[dynamicTableSize];

for (int i = 0; i < dynamicTableSize; i++)

identificators[i] = new List<string>();

}

public Token addIdentificator(string name)

{

var indexInTable = name.CalculateHash(dynamicTableSize);

var indexInRow = identificators[indexInTable].IndexOf(name);

if (indexInRow != -1)

return new Token(TokenType.identificator, indexInTable, indexInRow);

else

{

identificators[indexInTable].Add(name);

return new Token(TokenType.identificator, indexInTable, 0);

}

}

public Token addConstant(int value)

{

var indexOfConstant = constants.IndexOf(value);

if (indexOfConstant != -1)

return new Token(TokenType.constant, indexOfConstant, -1);

else

{

constants.Add(value);

return new Token(TokenType.constant, constants.Count - 1, -1);

}

}

}

}