Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**ЛАБОРАТОРНА РОБОТА № 1**

**з дисципліни**

**«Теорія формальних мов та компіляцій»**

**Тема: «Розробка лексичного аналізатора»**

Виконав:

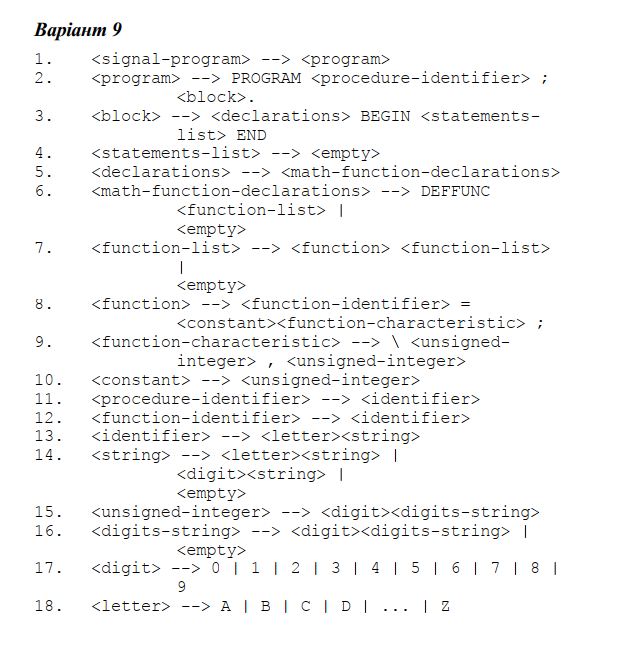
студента групи КП-91мп

Бабенко Валерій Павлович

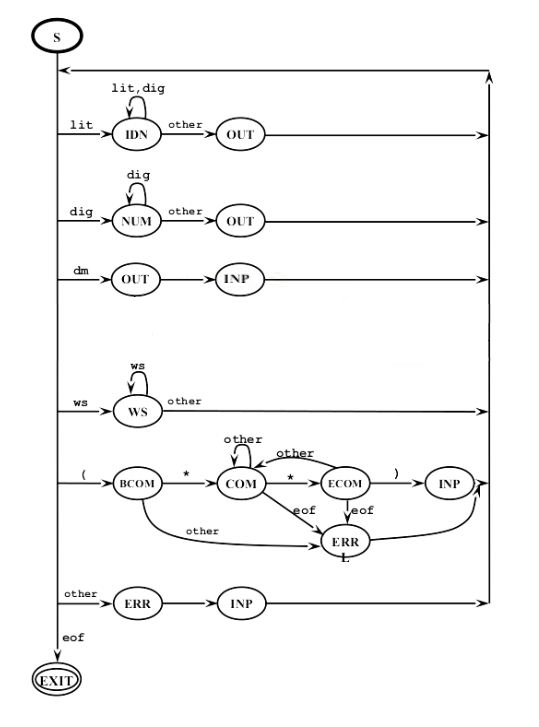
Київ 2019

**Постановка завдання:**

Розробити програму лексичного аналізатора (ЛА) для підмножини мови програмування SIGNAL.



**Граф автомату, що задає алгоритм ЛА**



**Лістинг**

*Файл Program.cs*

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Marchenko\_Lab1  {  class Program  {  static void Main(string[] args)  {  LexicAnalyst lexAnalyst = new LexicAnalyst("input.txt", "keywords.txt", "separs.txt");  lexAnalyst.analyze();  lexAnalyst.writeLexemStringToConsole();  Console.ReadKey();  }  }  } |

*Файл LexicAnalyst.cs*

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.IO;  namespace Marchenko\_Lab2  {  public enum State  {  Start, Input, Identifier, Constant, Delimiter, WhiteSpace, BeginComment, Comment,  EndComment  }  public enum LexemType  {  Identifier, Keyword, Delimiter, Constant  }  public struct Lexem  {  public int line;  public int column;  public int code;  public string value;  public LexemType type;  }  public struct Error  {  public int line;  public int column;  public string message;  public Error(int \_line, int \_column, string \_message)  {  line = \_line;  column = \_column;  message = \_message;  }  }  class LexicAnalyst  {  private const int MS = 501;  public int CurrentLine { get; protected set; }  public int CurrentColumn { get; protected set; }  public List<Lexem> LexemList { get; protected set; }  public List<Error> ErrorList { get; protected set; }  private State currentState;  private StreamReader programText;  private StringBuilder currentLexemString;  private Lexem nextLexem;  private int lastId, lastConst;  public Dictionary<string, int> Identifiers { get; protected set; }  public Dictionary<string, int> Constants { get; protected set; }  public Dictionary<string, int> Keywords { get; protected set; }  public Dictionary<string, int> Delimiters { get; protected set; }  public LexicAnalyst(string programTextFilePath, string keywordsTableFilePath, string separTableFilePath = "")  {  CurrentLine = 1;  lastId = 1001;  lastConst = 501;  Constants = new Dictionary<string, int>();  Keywords = new Dictionary<string, int>();  Identifiers = new Dictionary<string, int>();  Delimiters = new Dictionary<string, int>();  programText = new StreamReader(programTextFilePath);  ErrorList = new List<Error>();  LexemList = new List<Lexem>();  currentLexemString = new StringBuilder();  using (StreamReader keywordsTable = new StreamReader(keywordsTableFilePath))  {  while (!keywordsTable.EndOfStream)  {  string[] pair = keywordsTable.ReadLine().Trim().Split(new char[1] { ' ' });  Keywords.Add(pair[0], Convert.ToInt32(pair[1]));  }  }  using (StreamReader separatorTable = new StreamReader(separTableFilePath))  {  while (!separatorTable.EndOfStream)  {  string[] pair = separatorTable.ReadLine().Trim().Split(new char[1] { ' ' });  Delimiters.Add(pair[0], Convert.ToInt32(pair[1]));  }  }  currentState = State.Start;  }  public void analyze()  {  int? current = '\0';  while ((current = getNextChatacter()) != null)  {  CurrentColumn++;  if (current == '\r')  {  defineState((char?)current);  continue;  }  if (current == '\n')  {  CurrentColumn = 0;  CurrentLine++;  }  defineState((char?)current);  }  defineState((char?)current);  }  private char? getNextChatacter()  {  if (currentState == State.Start && programText.EndOfStream)  {  ErrorList.Add(new Error(0, 0, "File is empty!"));  return null;  }  else if (programText.EndOfStream)  return null;  else  return (char)programText.Read();  }  private void defineState(char? current)  {  if (current == null)  {  if (currentState == State.BeginComment || currentState == State.Comment || currentState == State.EndComment)  defineError('\0');  if (currentLexemString.Length > 0)  defineLexem();  return;  }  char symbol = (char)current;  switch (currentState)  {  case State.Start:  case State.Input:  if (Char.IsUpper(symbol))  {  nextLexem.line = CurrentLine;  nextLexem.column = CurrentColumn;  currentLexemString.Append(symbol);  currentState = State.Identifier;  return;  }  if (Char.IsDigit(symbol))  {  nextLexem.line = CurrentLine;  nextLexem.column = CurrentColumn;  currentLexemString.Append(symbol);  currentState = State.Constant;  return;  }  if (symbol == '(')  {  currentState = State.BeginComment;  return;  }  if (isSeparator(symbol))  {  goto case State.Delimiter;  }  if (isWhiteSpace(symbol))  {  return;  }  defineError(symbol);  return;  case State.Identifier:  if (Char.IsUpper(symbol) || Char.IsDigit(symbol))  {  currentLexemString.Append(symbol);  return;  }  if (isSeparator(symbol))  {  defineLexem();  goto case State.Delimiter;  }  if (isWhiteSpace(symbol))  {  if (currentLexemString.Length != 0)  defineLexem();  return;  }  defineError(symbol);  return;  case State.Constant:  if (Char.IsDigit(symbol))  {  currentLexemString.Append(symbol);  return;  }  if (isSeparator(symbol))  {  defineLexem();  goto case State.Delimiter;  }  if (isWhiteSpace(symbol))  {  if (currentLexemString.Length != 0)  defineLexem();  return;  }  defineError(symbol);  return;  case State.Delimiter:  currentLexemString.Append(symbol);  currentState = State.Delimiter;  defineLexem();  return;  case State.BeginComment:  if (symbol == '\*')  {  currentState = State.Comment;  return;  }  defineError(symbol);  return;  case State.Comment:  if (symbol == '\*')  currentState = State.EndComment;  return;  case State.EndComment:  if (symbol == ')')  {  currentState = State.Input; return;  }  currentState = State.Comment;  return;  }  }  private bool isWhiteSpace(char symbol)  {  int current = (int)symbol;  switch (current)  {  case 32:  case 13:  case 10:  case 9:  case 11:  case 12:  return true;  }  return false;  }  private bool isSeparator(char symbol)  {  foreach (var item in Delimiters)  if (Char.ToString(symbol) == item.Key)  return true;  return false;  }  private void defineError(char currentSymbol)  {  StringBuilder message = new StringBuilder("Unresolved symbol: " + currentSymbol + "\n line:" + CurrentLine + " column:" + CurrentColumn + "\n");  if (currentState == State.Identifier)  message.Append(" in identifier starts on [" + nextLexem.line + "," + nextLexem.column + "]");  if (currentState == State.Constant)  message.Append(" in unsigned integer starts on [" + nextLexem.line + "," + nextLexem.column + "]");  if (currentState == State.BeginComment)  {  message.Clear();  message.Append("Comment error. Asterisk expected!");  currentState = State.Input;  }  if (currentSymbol == '\0')  {  message.Clear();  message.Append("Comment error. Unclosed comment!");  }  ErrorList.Add(new Error(CurrentLine, CurrentColumn, message.ToString()));  }  private void defineLexem()  {  bool f = false;  switch (currentState)  {  case State.Identifier:  f = searchInTable(Keywords);  if (!f)  {  f = searchInTable(Identifiers);  nextLexem.type = LexemType.Identifier;  nextLexem.code = lastId++;  Identifiers.Add(currentLexemString.ToString(), nextLexem.code);  nextLexem.type = LexemType.Identifier;  }  else  {  nextLexem.type = LexemType.Keyword;  }  outputLexem();  break;  case State.Constant:  f = searchInTable(Constants);  if (!f)  {  nextLexem.code = lastConst++;  Constants.Add(currentLexemString.ToString(), nextLexem.code);  }  nextLexem.type = LexemType.Constant;  outputLexem();  break;  case State.Delimiter:  f = searchInTable(Delimiters);  nextLexem.type = LexemType.Delimiter;  nextLexem.line = CurrentLine;  nextLexem.column = CurrentColumn;  outputLexem();  break;  case State.WhiteSpace:  break;  }  }  private void outputLexem()  {  nextLexem.value = currentLexemString.ToString();  LexemList.Add(nextLexem);  currentLexemString.Clear();  currentState = State.Input;  return;  }  private bool searchInTable(Dictionary<string, int> dict)  {  foreach (var item in dict)  if (item.Key == currentLexemString.ToString())  {  nextLexem.code = item.Value;  return true;  }  return false;  }  private void displayTable(Dictionary<string, int> dict, string tableName)  {  Console.BackgroundColor = ConsoleColor.Magenta;  Console.ForegroundColor = ConsoleColor.White;  Console.WriteLine("\n{0}\tKey", tableName);  Console.BackgroundColor = ConsoleColor.Black;  Console.ForegroundColor = ConsoleColor.Gray;  foreach (var item in dict)  Console.WriteLine("{0}\t{1}", item.Key, item.Value);  }  public void display()  {  Console.BackgroundColor = ConsoleColor.Magenta;  Console.ForegroundColor = ConsoleColor.White;  Console.WriteLine("Value\tCode\tcoord.\tType");  Console.BackgroundColor = ConsoleColor.Black;  Console.ForegroundColor = ConsoleColor.Gray;  foreach (var item in LexemList)  {  Console.WriteLine(item.value + "\t" + item.code + "\t({0},{1})\t" + item.type, item.line, item.column);  }    //displayTable(Identifiers, "ident");  //displayTable(Keywords, "keyw");  //displayTable(Constants, "const");  //displayTable(Delimiters, "delim");  if (ErrorList.Count != 0)  Console.WriteLine("\nThe errors:");  foreach (var item in ErrorList)  {  Console.WriteLine(item.message);  }  }  public Dictionary<string, Dictionary<string, int>> getTables()  {  Dictionary<string, Dictionary<string, int>> Tables = new Dictionary<string, Dictionary<string, int>>();  Tables.Add("Identifiers", Identifiers);  Tables.Add("Keywords", Keywords);  Tables.Add("Delimiters", Delimiters);  Tables.Add("Constants", Constants);  return Tables;  }  public List<Lexem> getLexemList()  {  return this.LexemList;  }  }  } |

**Приклад роботи**

*Файл input.txt*

|  |
| --- |
| PROGRAM AAA;  (\* commentary \*)  DEFFUNC FUNC1=60\30,10; FUNC2 = 99\5,30;  BEGIN  END. |

*Вивід*

