Lab 4:

using System;

using System.Collections.Generic;

using System.Text.RegularExpressions;

class LexicalAnalyzer

{

// List of keywords

static List<string> keywordList = new List<string>() { "int", "float", "while", "main", "if", "else", "new" };

// Regular expressions for lexemes

static Regex variable\_Reg = new Regex(@"^[A-Za-z\_][A-Za-z0-9\_]\*$");

static Regex constants\_Reg = new Regex(@"^[0-9]+([.][0-9]+)?([e]([+|-])?[0-9]+)?$");

static Regex operators\_Reg = new Regex(@"^[-\*+/><&&||=]$");

static Regex special\_Reg = new Regex(@"^[.,'\[\]{}();:?]$");

static void Main()

{

// Prompt user for input

Console.WriteLine("Enter your code (press Enter twice to end input):");

string userInput = GetUserInput();

// Call the lexical analyzer function

LexicalAnalyzerFunction(userInput);

}

// Method to get multiline input from user

static string GetUserInput()

{

string input = "";

string line;

while (true)

{

line = Console.ReadLine();

// Break if the user presses Enter twice to stop input

if (string.IsNullOrWhiteSpace(line))

break;

input += line + "\n";

}

return input;

}

static void LexicalAnalyzerFunction(string input)

{

// Initialize symbol table

List<string[]> symbolTable = new List<string[]>();

// Split input into lines

string[] lines = input.Split('\n');

// Output buffers

List<string> tokens = new List<string>();

// Process each line in the input

foreach (var line in lines)

{

string currentToken = "";

// Loop through each character in the line

for (int i = 0; i < line.Length; i++)

{

char ch = line[i];

// Check for a valid lexeme (a token)

if (char.IsWhiteSpace(ch) || special\_Reg.IsMatch(ch.ToString()))

{

if (!string.IsNullOrEmpty(currentToken))

{

ProcessToken(currentToken, tokens, symbolTable);

currentToken = "";

}

if (special\_Reg.IsMatch(ch.ToString()))

{

tokens.Add($"<punc, {ch}>");

}

}

else

{

currentToken += ch;

}

}

// Process the last token in the line

if (!string.IsNullOrEmpty(currentToken))

{

ProcessToken(currentToken, tokens, symbolTable);

}

}

// Display tokens

Console.WriteLine("Tokens:");

foreach (var token in tokens)

{

Console.WriteLine(token);

}

// Display symbol table

Console.WriteLine("\nSymbol Table:");

foreach (var entry in symbolTable)

{

Console.WriteLine($"Index: {entry[0]}, Name: {entry[1]}, Type: {entry[2]}, Value: {entry[3]}");

}

}

static void ProcessToken(string token, List<string> tokens, List<string[]> symbolTable)

{

// Match the token with regex patterns

if (keywordList.Contains(token))

{

tokens.Add($"<keyword, {token}>");

}

else if (variable\_Reg.IsMatch(token))

{

tokens.Add($"<var, {token}>");

// Check if the variable is already in the symbol table

if (!symbolTable.Exists(entry => entry[1] == token))

{

// Assuming "int" as the default type for simplicity

symbolTable.Add(new string[] { symbolTable.Count.ToString(), token, "int", "Unknown" });

}

}

else if (constants\_Reg.IsMatch(token))

{

tokens.Add($"<constant, {token}>");

// If constant is a number, we need to handle its type (int or float)

string type = token.Contains(".") ? "float" : "int";

// Add constant to the symbol table (with proper type and value)

if (!symbolTable.Exists(entry => entry[1] == token))

{

symbolTable.Add(new string[] { symbolTable.Count.ToString(), token, type, token });

}

}

else if (operators\_Reg.IsMatch(token))

{

tokens.Add($"<operator, {token}>");

}

else

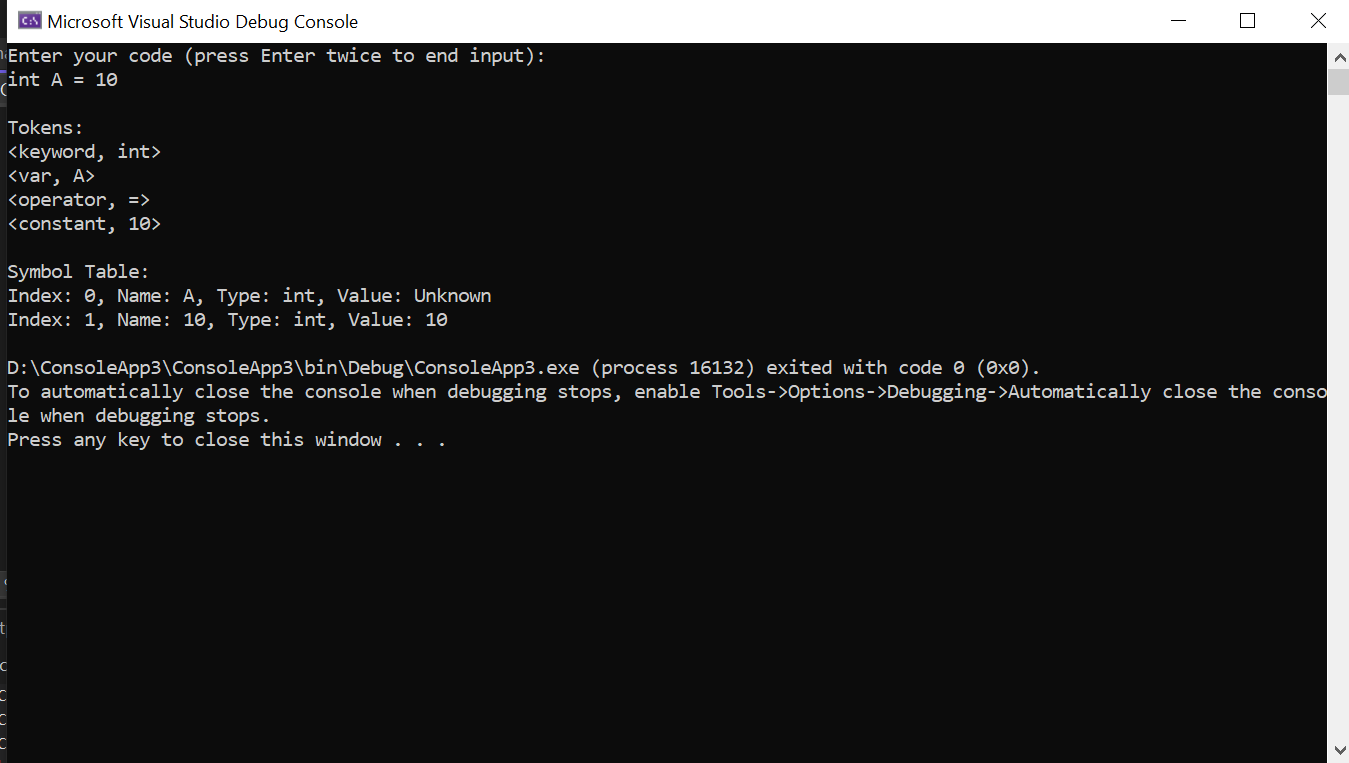
{

Console.WriteLine($"Unknown token: {token}");

}

}

}



Lab 5:

using System;

using System.Collections.Generic;

using System.Text.RegularExpressions;

class LexicalAnalyzer

{

// List of keywords

static List<string> keywordList = new List<string>() { "int", "float", "while", "main", "if", "else", "new" };

// Regular expressions for lexemes

static Regex variable\_Reg = new Regex(@"^[A-Za-z\_][A-Za-z0-9\_]\*$");

static Regex constants\_Reg = new Regex(@"^[0-9]+([.][0-9]+)?([e]([+|-])?[0-9]+)?$");

static Regex operators\_Reg = new Regex(@"^[-\*+/><&&||=]$");

static Regex special\_Reg = new Regex(@"^[.,'\[\]{}();:?]$");

static void Main()

{

// Prompt user for input

Console.WriteLine("Enter your code :");

string userInput = GetUserInput();

// Call the lexical analyzer function

LexicalAnalyzerFunction(userInput);

}

// Method to get multiline input from user

static string GetUserInput()

{

string input = "";

string line;

while (true)

{

line = Console.ReadLine();

// Break if the user presses Enter twice to stop input

if (string.IsNullOrWhiteSpace(line))

break;

input += line + "\n";

}

return input;

}

static void LexicalAnalyzerFunction(string input)

{

// Initialize symbol table using a Dictionary (hash table)

Dictionary<string, string[]> symbolTable = new Dictionary<string, string[]>();

// Split input into lines

string[] lines = input.Split('\n');

// Output buffers

List<string> tokens = new List<string>();

// Process each line in the input

foreach (var line in lines)

{

string currentToken = "";

// Loop through each character in the line

for (int i = 0; i < line.Length; i++)

{

char ch = line[i];

// Check for a valid lexeme (a token)

if (char.IsWhiteSpace(ch) || special\_Reg.IsMatch(ch.ToString()))

{

if (!string.IsNullOrEmpty(currentToken))

{

ProcessToken(currentToken, tokens, symbolTable);

currentToken = "";

}

if (special\_Reg.IsMatch(ch.ToString()))

{

tokens.Add($"<punc, {ch}>");

}

}

else

{

currentToken += ch;

}

}

// Process the last token in the line

if (!string.IsNullOrEmpty(currentToken))

{

ProcessToken(currentToken, tokens, symbolTable);

}

}

// Display tokens

Console.WriteLine("Tokens:");

foreach (var token in tokens)

{

Console.WriteLine(token);

}

// Display symbol table

Console.WriteLine("\nSymbol Table:");

foreach (var entry in symbolTable)

{

string[] symbolData = entry.Value;

Console.WriteLine($"Index: {symbolData[0]}, Name: {symbolData[1]}, Type: {symbolData[2]}, Value: {symbolData[3]}");

}

}

static void ProcessToken(string token, List<string> tokens, Dictionary<string, string[]> symbolTable)

{

// Match the token with regex patterns

if (keywordList.Contains(token))

{

tokens.Add($"<keyword, {token}>");

}

else if (variable\_Reg.IsMatch(token))

{

tokens.Add($"<var, {token}>");

// Check if the variable is already in the symbol table

if (!symbolTable.ContainsKey(token))

{

// Assuming "int" as the default type for simplicity

symbolTable.Add(token, new string[] { symbolTable.Count.ToString(), token, "int", "Unknown" });

}

}

else if (constants\_Reg.IsMatch(token))

{

tokens.Add($"<constant, {token}>");

// If constant is a number, we need to handle its type (int or float)

string type = token.Contains(".") ? "float" : "int";

// Add constant to the symbol table (with proper type and value)

if (!symbolTable.ContainsKey(token))

{

symbolTable.Add(token, new string[] { symbolTable.Count.ToString(), token, type, token });

}

}

else if (operators\_Reg.IsMatch(token))

{

tokens.Add($"<operator, {token}>");

}

else

{

Console.WriteLine($"Unknown token: {token}");

}

}

}

