**Muhammad Usman (sp22-bcs-036)**

**Lab 1: task**

using System;

using System.Text;

using System.Text.RegularExpressions;

class Program

{

static void Main(string[] args)

{

// Sample inputs

Console.WriteLine("Enter your first name: ");

string firstName = Console.ReadLine();

Console.WriteLine("Enter your last name: ");

string lastName = Console.ReadLine();

Console.WriteLine("Enter your registration number: ");

string regNumber = Console.ReadLine();

Console.WriteLine("Enter your favorite food: ");

string food = Console.ReadLine();

Console.WriteLine("Enter your favorite game: ");

string game = Console.ReadLine();

// Generate the password

string password = GeneratePassword(firstName, lastName, regNumber, food, game);

// Display the generated password

Console.WriteLine("Generated Password: " + password);

}

static string GeneratePassword(string firstName, string lastName, string regNumber, string food, string game)

{

// Combine all input values

string combined = firstName + lastName + regNumber + food + game;

// Regular expression to remove any unwanted characters (non-alphanumeric)

string sanitized = Regex.Replace(combined, @"[^a-zA-Z0-9]", "");

// Make the string more complex by adding special characters and digits

string complexPassword = sanitized;

// Add some random numbers and special characters

Random rand = new Random();

string specialChars = "!@#$%^&\*()\_+[]{}|;:,.<>?/~`";

for (int i = 0; i < 4; i++)

{

// Add random number

complexPassword += rand.Next(0, 10).ToString();

// Add random special character

complexPassword += specialChars[rand.Next(specialChars.Length)];

}

// Ensure password length is at least 12 characters

if (complexPassword.Length < 12)

{

complexPassword = complexPassword.PadLeft(12, 'X'); // Add filler 'X' if too short

}

// Randomly shuffle the password to increase complexity

StringBuilder shuffledPassword = new StringBuilder();

while (complexPassword.Length > 0)

{

int index = rand.Next(complexPassword.Length);

shuffledPassword.Append(complexPassword[index]);

complexPassword = complexPassword.Remove(index, 1);

}

return shuffledPassword.ToString();

}

}



**Lab 1: task 2**

using System;

using System.Text.RegularExpressions;

public class Program

{

public static void Main()

{

// Hardcoded password for validation

string password = "Sp22-bcs-036"; // Example password

// Regular expression pattern for the requirements

string pattern = @"^(?=(.\*\d.\*){2})(?=.\*[A-Z])(?=(.\*[a-z]){4})(?=(.\*[-!@#$%^&\*(),.?\""{}|<>]){2}).{1,12}$";

// Check if the password matches the pattern

if (Regex.IsMatch(password, pattern))

{

Console.WriteLine("Password is valid.");

}

else

{

Console.WriteLine("Password is invalid.");

}

}

}



**lab 2: task 1**

using System;

using System.Text.RegularExpressions;

class Program

{

static void Main()

{

// The regular expression for logical operators and parentheses

string pattern = @"\s\*(&&|\|\||!|\(|\))\s\*";

// Test string with logical operators and parentheses

string input = "x && y || !z (x || y)";

// Create a Regex object with the pattern

Regex regex = new Regex(pattern);

// Find all matches

MatchCollection matches = regex.Matches(input);

// Output the matches

foreach (Match match in matches)

{

Console.WriteLine($"Found: {match.Value}");

}

}

}



**lab 2:task 2**

using System;

using System.Text.RegularExpressions;

class Program

{

static void Main()

{

// The regular expression for relational operators

string pattern = @"\s\*(==|!=|>=|<=|>|<)\s\*";

// Test string with relational operators

string input = "a == b && c != d || e >= f && g < h";

// Create a Regex object with the pattern

Regex regex = new Regex(pattern);

// Find all matches

MatchCollection matches = regex.Matches(input);

// Output the matches

foreach (Match match in matches)

{

Console.WriteLine($"Found: {match.Value}");

}

}

}



**Lab 3: task 1**

using System;

using System.Text.RegularExpressions;

class Program

{

static void Main()

{

// Regular expression for floating point numbers with length <= 6

string pattern = @"^[+-]?\d{1,3}(\.\d{1,3})?$|^[+-]?\.\d{1,3}$";

// Test strings

string[] testStrings = {

"123", // valid

"-12.34", // valid

"+0.567", // valid

".678", // valid

"0.5", // valid

"123456", // invalid

"1.2345", // invalid

"+1234", // invalid

".1234" // invalid

};

// Check each string against the regex

foreach (var test in testStrings)

{

bool isMatch = Regex.IsMatch(test, pattern);

Console.WriteLine($"{test}: {(isMatch ? "Valid" : "Invalid")}");

}

}

}



**Lab 4: task 1**

using System;

class LexicalAnalyzer

{

const int BUFFER\_SIZE = 1024;

const int KEYWORD\_COUNT = 3;

static string[] keywords = { "int", "if", "else" };

static char[] buffer = new char[BUFFER\_SIZE];

static int bufferIndex = 0;

static bool IsKeyword(string lexeme)

{

for (int i = 0; i < KEYWORD\_COUNT; i++)

{

if (lexeme.Equals(keywords[i]))

return true;

}

return false;

}

static void LexicalAnalyzerFunc()

{

string lexeme = "";

while (bufferIndex < buffer.Length && buffer[bufferIndex] != '\0')

{

char currentChar = buffer[bufferIndex];

if (char.IsWhiteSpace(currentChar))

{

bufferIndex++;

continue;

}

lexeme = "";

if (char.IsLetter(currentChar)) // Identifier or Keyword

{

while (bufferIndex < buffer.Length && (char.IsLetterOrDigit(buffer[bufferIndex])))

{

lexeme += buffer[bufferIndex];

bufferIndex++;

}

if (IsKeyword(lexeme))

Console.WriteLine($"Keyword: {lexeme}");

else

Console.WriteLine($"Identifier: {lexeme}");

}

else if (char.IsDigit(currentChar)) // Number

{

while (bufferIndex < buffer.Length && char.IsDigit(buffer[bufferIndex]))

{

lexeme += buffer[bufferIndex];

bufferIndex++;

}

Console.WriteLine($"Number: {lexeme}");

}

else // Operator or special character

{

Console.WriteLine($"Operator: {currentChar}");

bufferIndex++;

}

}

}

static void Main()

{

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

string input = Console.ReadLine();

buffer = input.ToCharArray();

LexicalAnalyzerFunc();

}

}

\

**Lab 5: task 1**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TABLE\_SIZE 10 // Hash table size

// Structure for a symbol table entry

typedef struct Symbol {

char name[50]; // Identifier name

char type[20]; // Data type (e.g., int, float)

int scope; // Scope level

struct Symbol \*next; // Pointer for chaining (linked list)

} Symbol;

// Hash table (Array of pointers to Symbol nodes)

Symbol \*symbolTable[TABLE\_SIZE];

// Hash function (Sum of ASCII values modulo table size)

int hashFunction(char \*name) {

int sum = 0;

for (int i = 0; name[i] != '\0'; i++) {

sum += name[i];

}

return sum % TABLE\_SIZE;

}

// Insert a symbol into the table

void insertSymbol(char \*name, char \*type, int scope) {

int index = hashFunction(name);

// Create a new symbol node

Symbol \*newSymbol = (Symbol \*)malloc(sizeof(Symbol));

strcpy(newSymbol->name, name);

strcpy(newSymbol->type, type);

newSymbol->scope = scope;

newSymbol->next = NULL;

// Insert at the beginning of the linked list (chaining)

if (symbolTable[index] == NULL) {

symbolTable[index] = newSymbol;

} else {

newSymbol->next = symbolTable[index];

symbolTable[index] = newSymbol;

}

printf("Inserted: %s (%s, scope: %d)\n", name, type, scope);

}

// Search for a symbol in the table

Symbol\* searchSymbol(char \*name) {

int index = hashFunction(name);

Symbol \*temp = symbolTable[index];

while (temp != NULL) {

if (strcmp(temp->name, name) == 0) {

return temp; // Found

}

temp = temp->next;

}

return NULL; // Not found

}

// Display the symbol table

void displaySymbolTable() {

printf("\nSymbol Table:\n");

printf("--------------------------------\n");

printf("| Index | Name | Type | Scope |\n");

printf("--------------------------------\n");

for (int i = 0; i < TABLE\_SIZE; i++) {

Symbol \*temp = symbolTable[i];

while (temp != NULL) {

printf("| %5d | %-7s | %-6s | %5d |\n", i, temp->name, temp->type, temp->scope);

temp = temp->next;

}

}

printf("--------------------------------\n");

}

// Main function for testing

int main() {

// Initializing the symbol table with NULL

for (int i = 0; i < TABLE\_SIZE; i++) {

symbolTable[i] = NULL;

}

// Insert some symbols

insertSymbol("x", "int", 1);

insertSymbol("y", "float", 1);

insertSymbol("sum", "int", 2);

insertSymbol("product", "int", 2);

insertSymbol("y", "char", 3); // Different scope

// Search for a symbol

char searchName[50];

printf("\nEnter variable name to search: ");

scanf("%s", searchName);

Symbol \*result = searchSymbol(searchName);

if (result) {

printf("Found: %s (%s, scope: %d)\n", result->name, result->type, result->scope);

} else {

printf("Symbol not found.\n");

}

// Display the symbol table

displaySymbolTable();

return 0;

}

**LAB 5: TASK 1**

using System;

using System.Collections.Generic;

class Symbol

{

public string Name { get; set; }

public string Type { get; set; }

public int Scope { get; set; }

public Symbol Next { get; set; } // Linked list chain

}

class SymbolTable

{

private const int TABLE\_SIZE = 10;

private List<Symbol>[] symbolTable;

public SymbolTable()

{

symbolTable = new List<Symbol>[TABLE\_SIZE];

for (int i = 0; i < TABLE\_SIZE; i++)

{

symbolTable[i] = new List<Symbol>();

}

}

// Hash function (Sum of ASCII values modulo table size)

private int HashFunction(string name)

{

int sum = 0;

foreach (char c in name)

{

sum += c;

}

return sum % TABLE\_SIZE;

}

// Insert a symbol into the table

public void InsertSymbol(string name, string type, int scope)

{

int index = HashFunction(name);

// Create a new symbol

Symbol newSymbol = new Symbol

{

Name = name,

Type = type,

Scope = scope,

Next = null

};

// Insert at the beginning of the linked list (chaining)

symbolTable[index].Add(newSymbol);

Console.WriteLine($"Inserted: {name} ({type}, scope: {scope})");

}

// Search for a symbol in the table

public Symbol SearchSymbol(string name)

{

int index = HashFunction(name);

foreach (var symbol in symbolTable[index])

{

if (symbol.Name.Equals(name, StringComparison.Ordinal))

{

return symbol; // Found

}

}

return null; // Not found

}

// Display the symbol table

public void DisplaySymbolTable()

{

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

Console.WriteLine("--------------------------------");

Console.WriteLine("| Index | Name | Type | Scope |");

Console.WriteLine("--------------------------------");

for (int i = 0; i < TABLE\_SIZE; i++)

{

foreach (var symbol in symbolTable[i])

{

Console.WriteLine($"| {i,5} | {symbol.Name,-7} | {symbol.Type,-6} | {symbol.Scope,5} |");

}

}

Console.WriteLine("--------------------------------");

}

}

class Program

{

static void Main()

{

SymbolTable table = new SymbolTable();

// Insert some symbols

table.InsertSymbol("x", "int", 1);

table.InsertSymbol("y", "float", 1);

table.InsertSymbol("sum", "int", 2);

table.InsertSymbol("product", "int", 2);

table.InsertSymbol("y", "char", 3); // Different scope

// Search for a symbol

Console.Write("\nEnter variable name to search: ");

string searchName = Console.ReadLine();

Symbol result = table.SearchSymbol(searchName);

if (result != null)

{

Console.WriteLine($"Found: {result.Name} ({result.Type}, scope: {result.Scope})");

}

else

{

Console.WriteLine("Symbol not found.");

}

// Display the symbol table

table.DisplaySymbolTable();

}

}  
