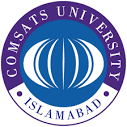
****

**Lab Mid Exam**

**Name: Sana Saeed**

**Reg No: Sp21-bcs-032**

**Course: Compiler construction**

**Date: 06/04/2024**

**Submitted To:**

**Mr. Syed Bilal Haider**

**QUESTION NO 1:**

Briefly describe the regex library of C#

**Answer:**

The theoretical aspects of regular expressions and how they are implemented in the C# Regex library:

**Regular Expressions (Regex):**

* Regular expressions are a powerful tool for pattern matching and string manipulation.
* They provide a concise and flexible means for describing text patterns.
* Regex patterns consist of a combination of literal characters, metacharacters, and quantifiers.
* Metacharacters such as ^, $, ., \*, +, ?, [, ], {, }, (, ), \, etc., have special meanings within a regex pattern.
* Quantifiers like \*, +, ?, {n}, {n,}, {n,m} specify the number of occurrences of a preceding element.

**Regex Class in C#:**

* The Regex class in C# encapsulates a compiled representation of a regular expression pattern.
* It provides methods for pattern matching, searching, replacing, and splitting strings based on regex patterns.
* The System.Text.RegularExpressions namespace contains this class and related types.
* The RegexOptions enum allows specifying various options like case insensitivity, multiline mode, and more.
* Example:

Regex regex = new Regex(@"\b\d{3}\b", RegexOptions.IgnoreCase);

**Match and MatchCollection:**

* The Match class represents a single match of a regex pattern within a string.
* It provides properties to access the matched value and captured groups within the match.
* The MatchCollection class represents a collection of matches found within a string.
* It allows iterating over multiple matches found in a single string.

**Groups and Capturing:**

* Parentheses () in a regex pattern create capturing groups.
* Capturing groups allow extracting portions of the matched string.
* Each capturing group can be accessed using the Groups property of a Match object.

**Constructors:**

The Regex class has various constructors for initializing instances with different parameters.

**Example:**

* Regex regex1 = new Regex(@"\b[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b", RegexOptions.IgnoreCase);
* Regex regex2 = new Regex(@"\b[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b", RegexOptions.IgnoreCase | RegexOptions.Compiled);

**Properties:**

* **Options:** Gets the options passed into the Regex constructor.
* **MatchTimeout:** Gets the time-out interval for pattern matching.

**Example:**

* Regex regex = new Regex(@"\b\d{3}\b", RegexOptions.IgnoreCase);
* RegexOptions options = regex.Options; // Returns RegexOptions.IgnoreCase

**Methods:**

* **Match:** Searches the specified input string for the first occurrence of the regular expression.
* **Matches:** Searches the specified input string for all occurrences of the regular expression.
* **Replace:** Replaces all occurrences of the regular expression pattern with a specified replacement string.

**Example:**

string input = "The price is $100 and $200.";

string pattern = @"\$\d+";

string result = Regex.Replace(input, pattern, "[$&]");

// Output: "The price is [$100] and [$200]."

**Performance Considerations:**

* Regular expressions can be computationally expensive, especially for complex patterns or large input strings.
* Compiling a regex pattern can improve performance for multiple uses of the same pattern.
* The RegexOptions.Compiled option compiles the regex pattern for improved performance.

**Use Cases:**

* **Validation:** Validate email addresses, phone numbers, etc.
* **Manipulation:** Replace or extract substrings based on patterns.
* **Search and Extraction:** Find and extract specific information from text.

Regular expressions are powerful tools for string manipulation and text processing tasks, and the C# Regex class provides a convenient way to work with them in your applications.

**Question No 02:**

Make recursive descent or LL1 parser or recursive descent parser for the following grammar:

S -> X$

X -> X % Y |Y

Y -> Y & Z |Z

Z -> k X k | g

**Answer:**

Making the above grammar right recursive to implement recursive descent or LL1 parser or recursive descent parser Hence, resulted grammar is

**CFG:**

**Right Recursive Grammer:**

S -> X$

X -> YX'

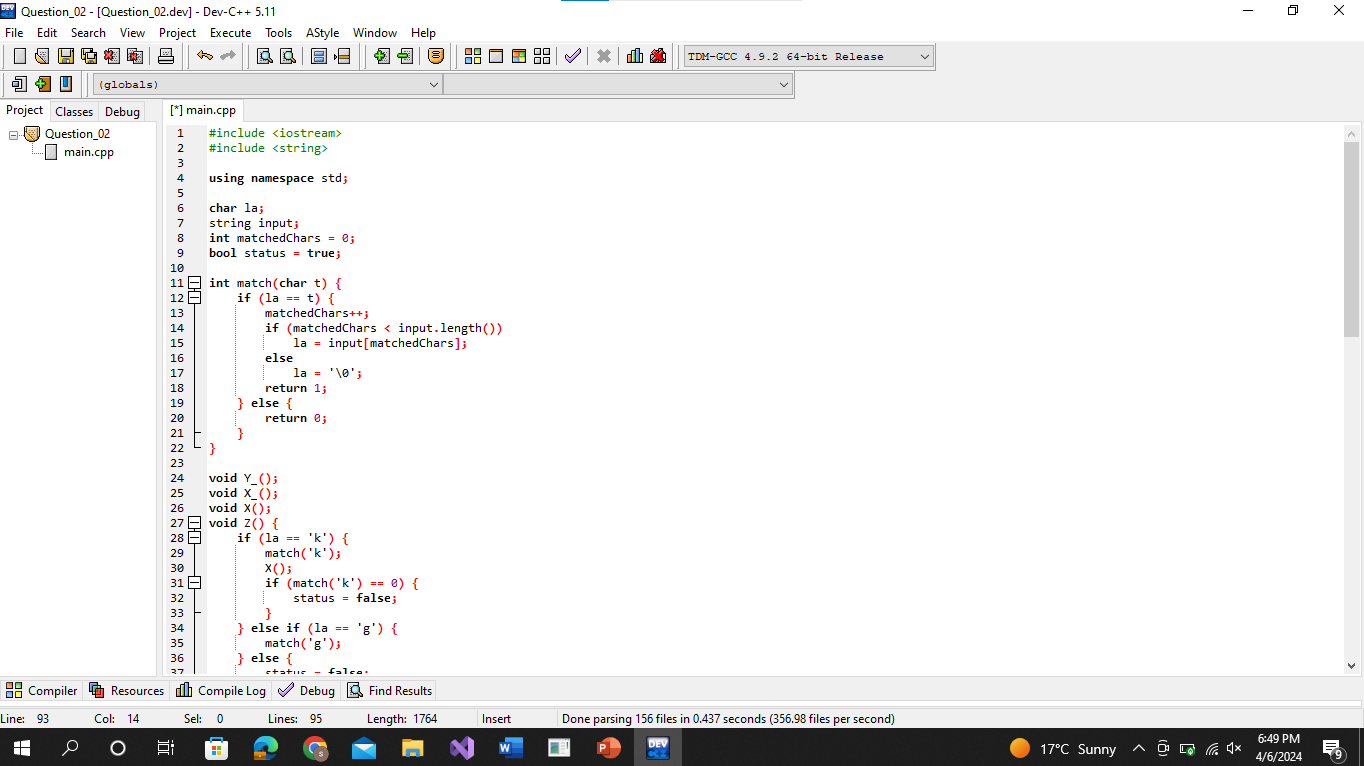
X' -> %YX' | ε

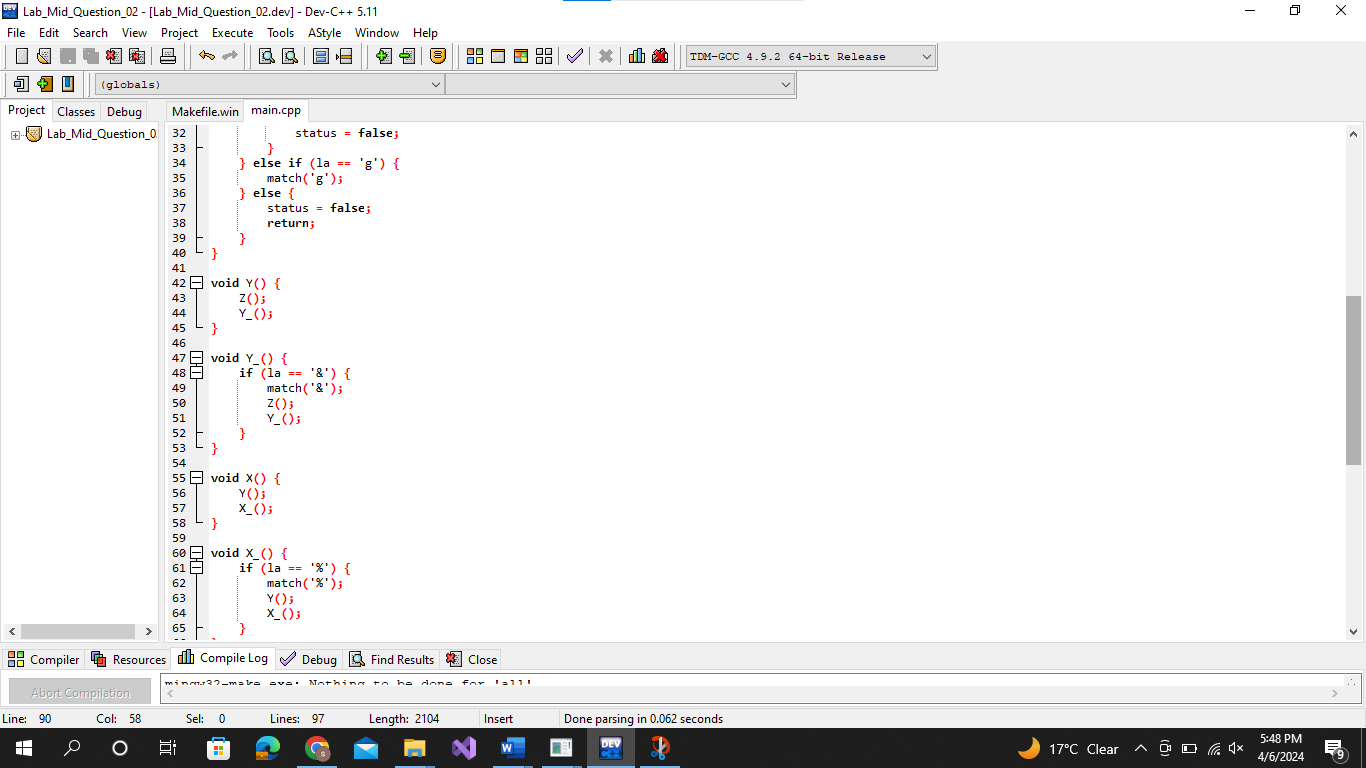
Y -> ZY'

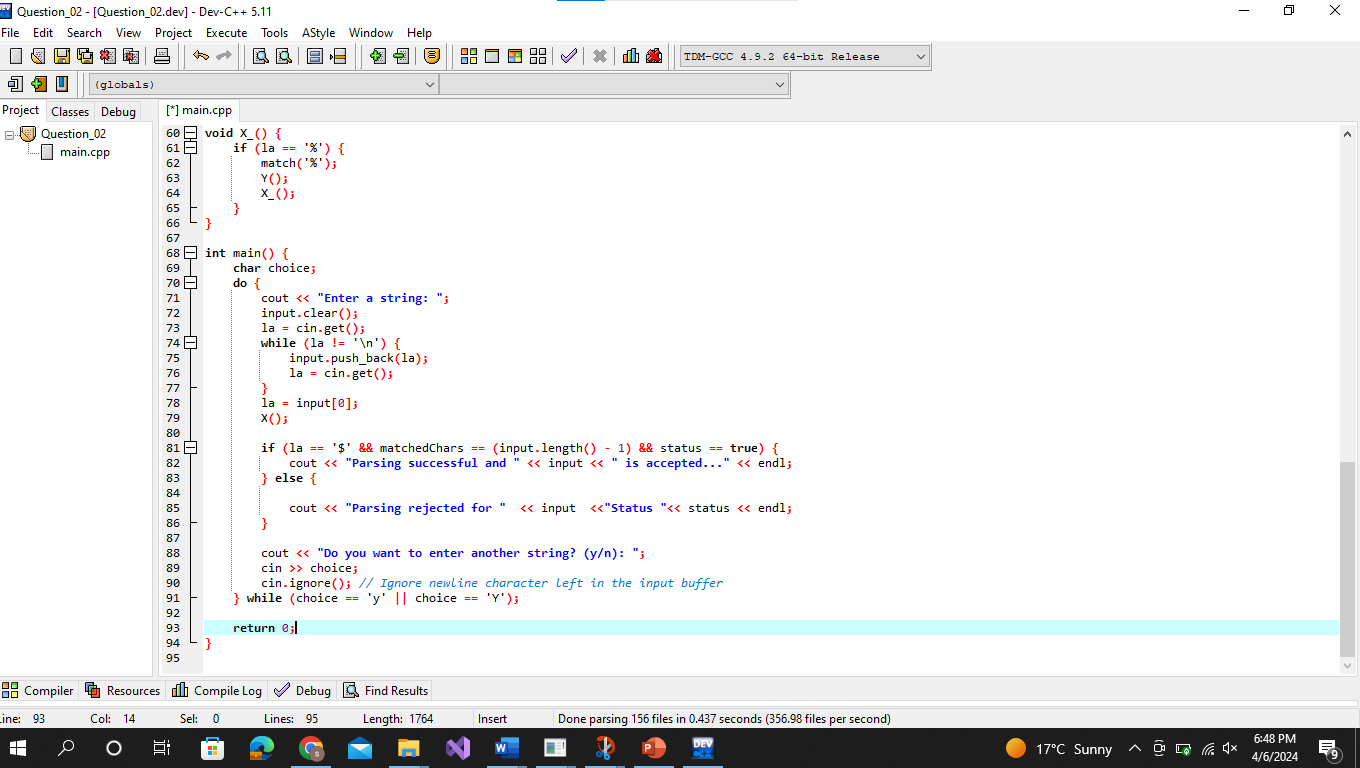
Y' -> &ZY' | ε

Z -> kXk | g

**Code:**

****

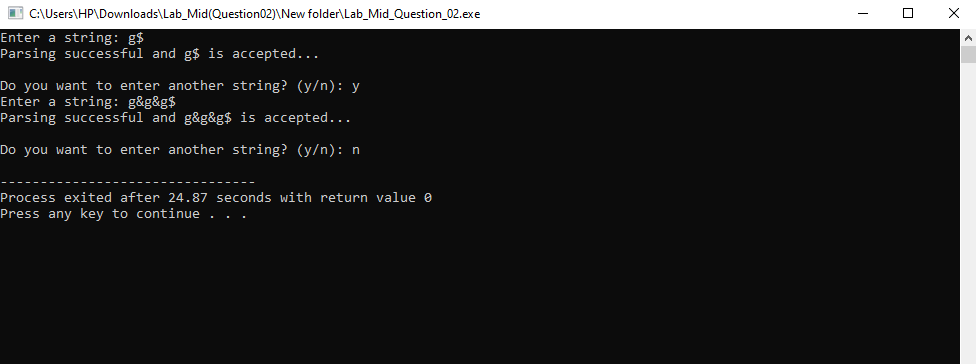
****

****

**Output:**

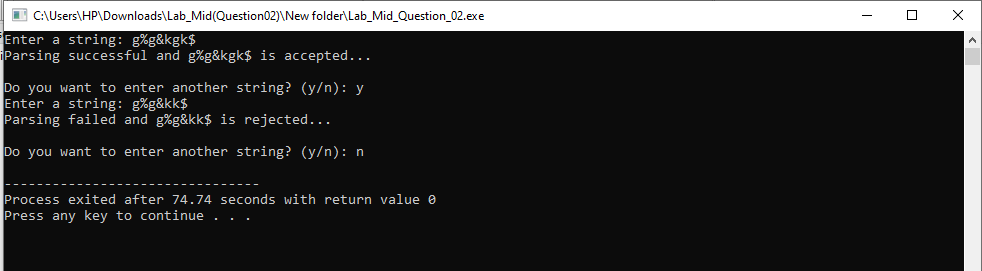
The are the inputs that are checked:

* g$
* g&g&g$

****

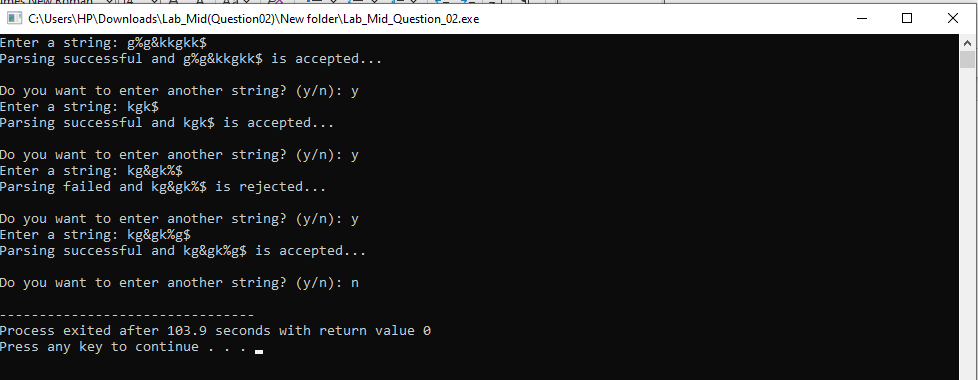
The are the inputs that are checked:

* g%g&kgk$
* g%g&kk$

****

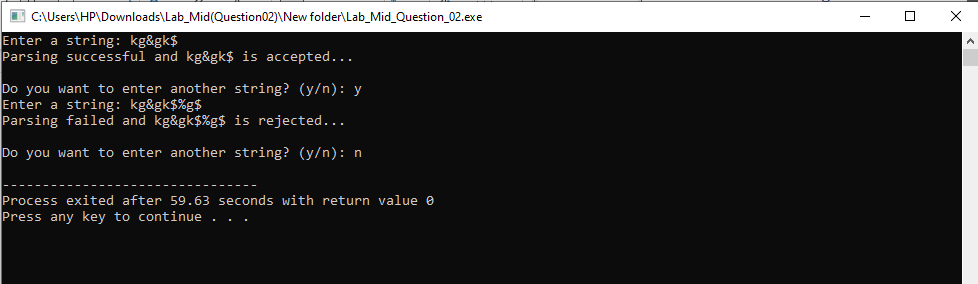
The are the inputs that are checked:

* g%g&kkgkk$
* kgk$
* kg&gk%$
* kg&gk%g$

****

The are the inputs that are checked:

* kg&gk$
* kg&gk$%g$

****

**Question No 03:**

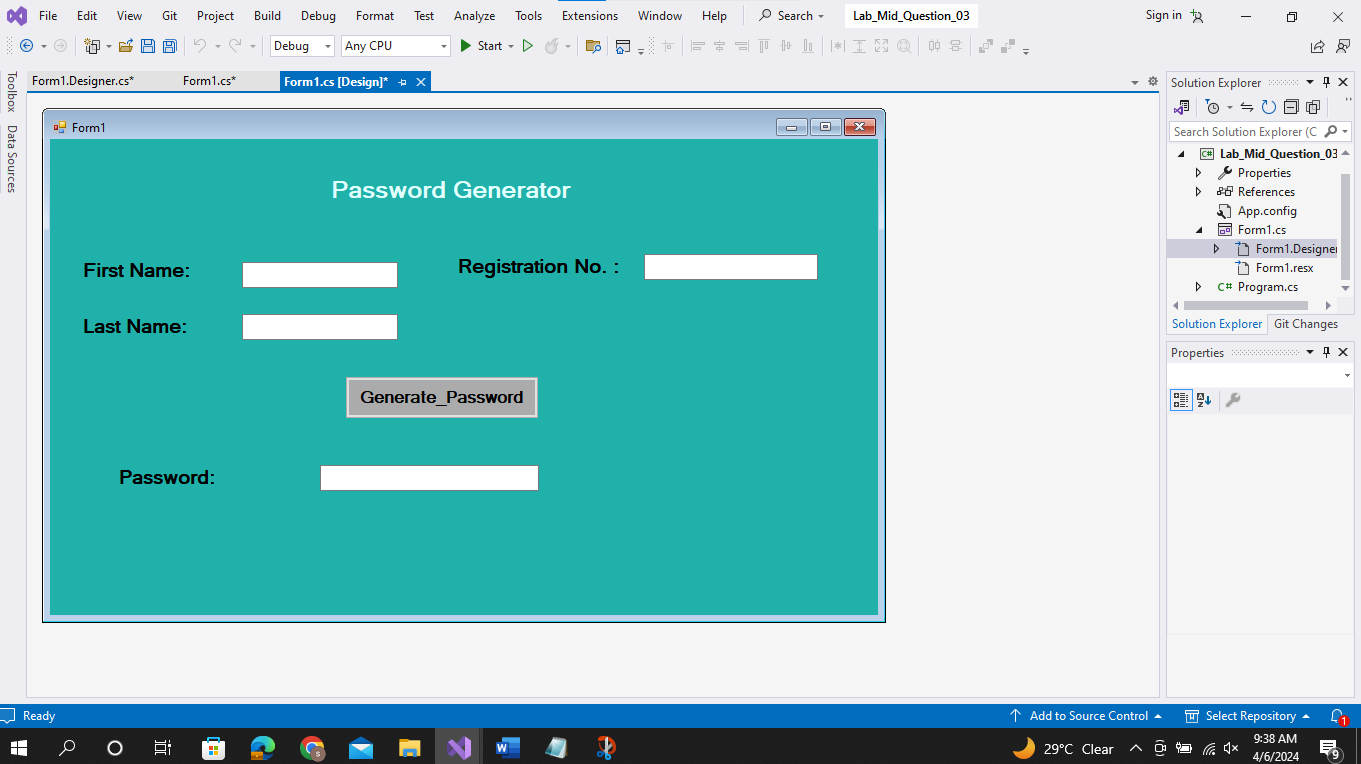
Make a Password generator according the following rules:

1. Atleast one uppercase alphabet
2. Atleast 4 numbers , two numbers must be your registration numbers
3. Atleast 2 special characters
4. Must contain initials of first and last name
5. Must contain all odd letters of your first name.
6. Must contain all even letters of your last name.
7. maximum length of 16

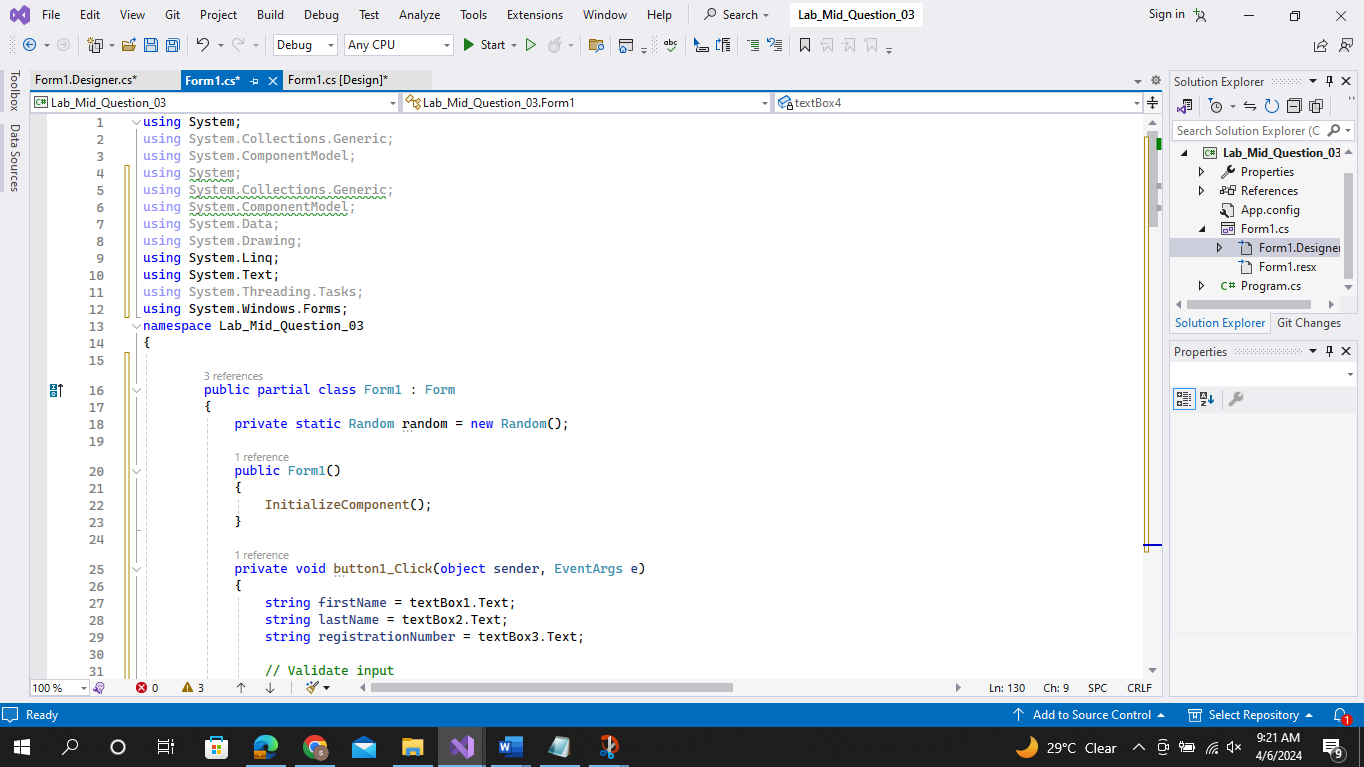
**Answer:**

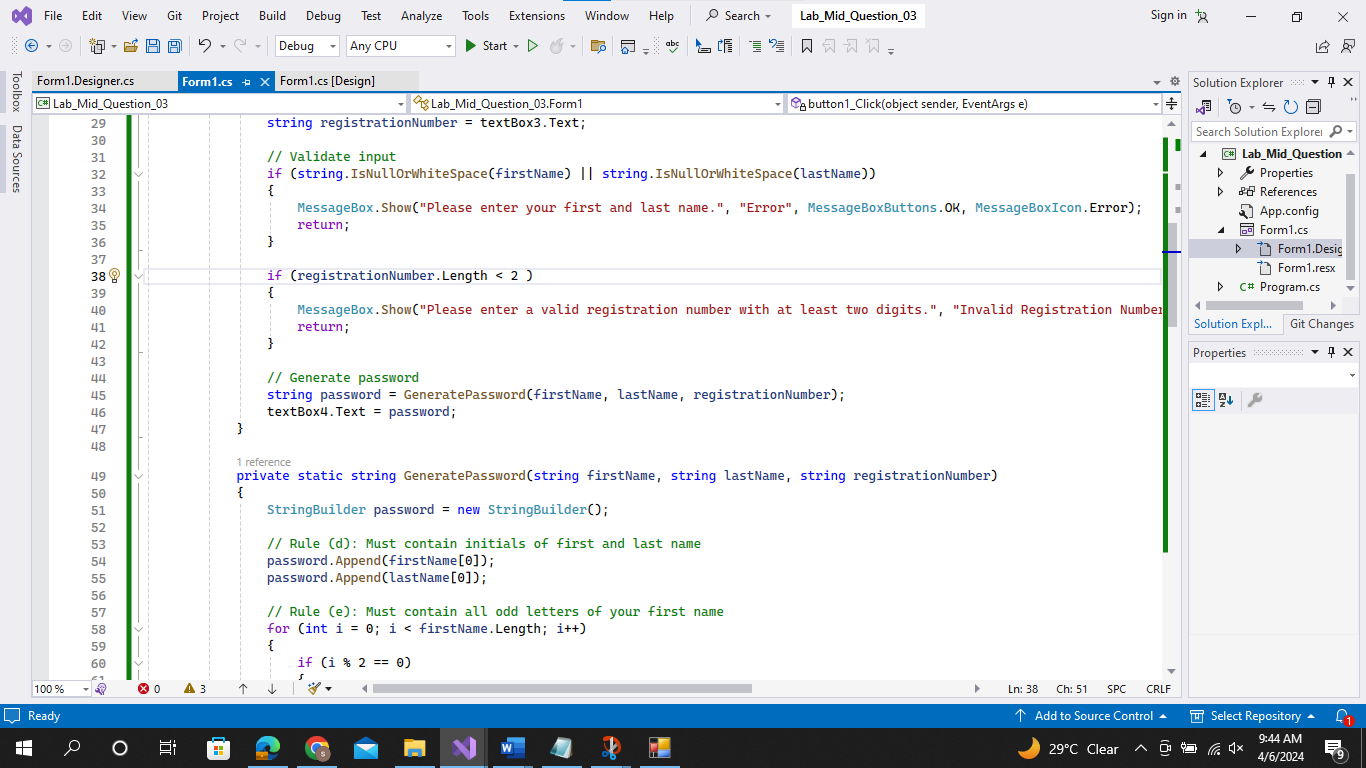
**It’s a Windows Form Application Hence,**

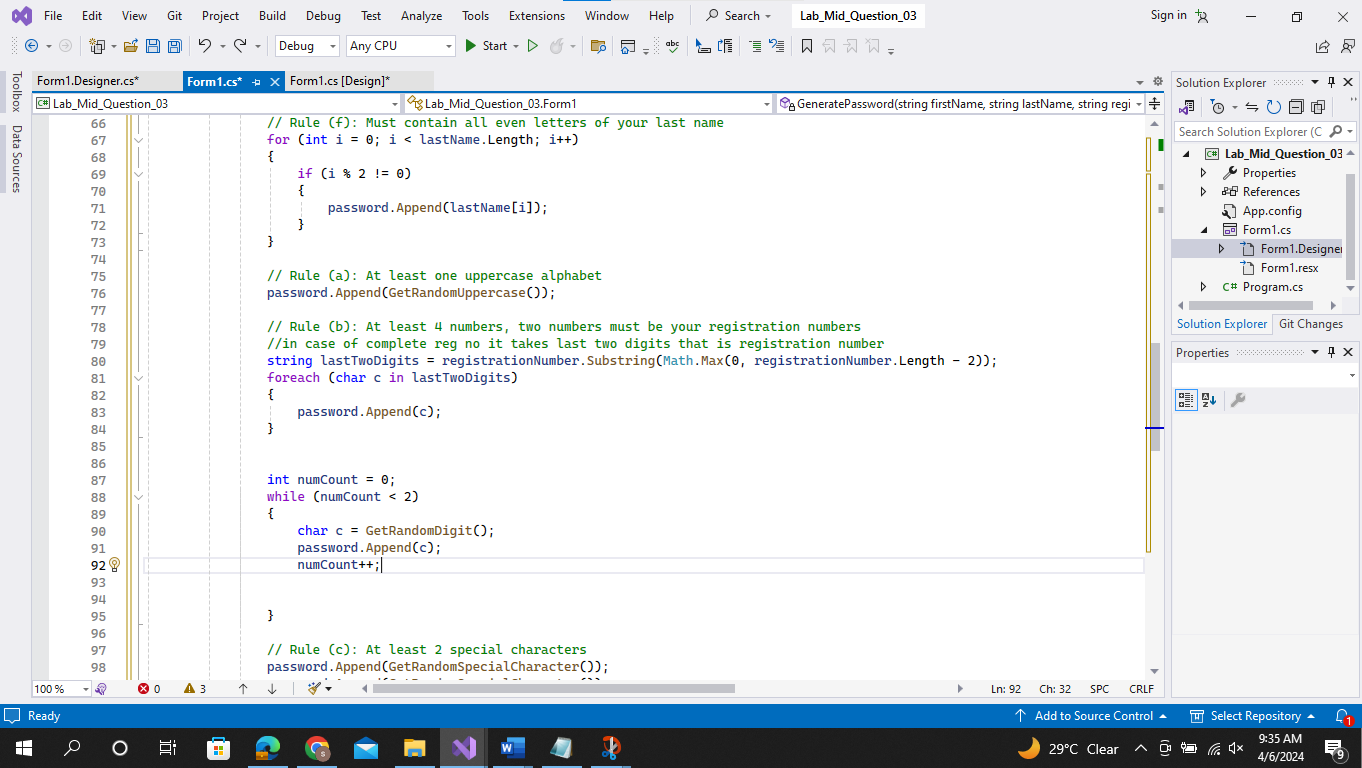
**Form(Design).cs:**

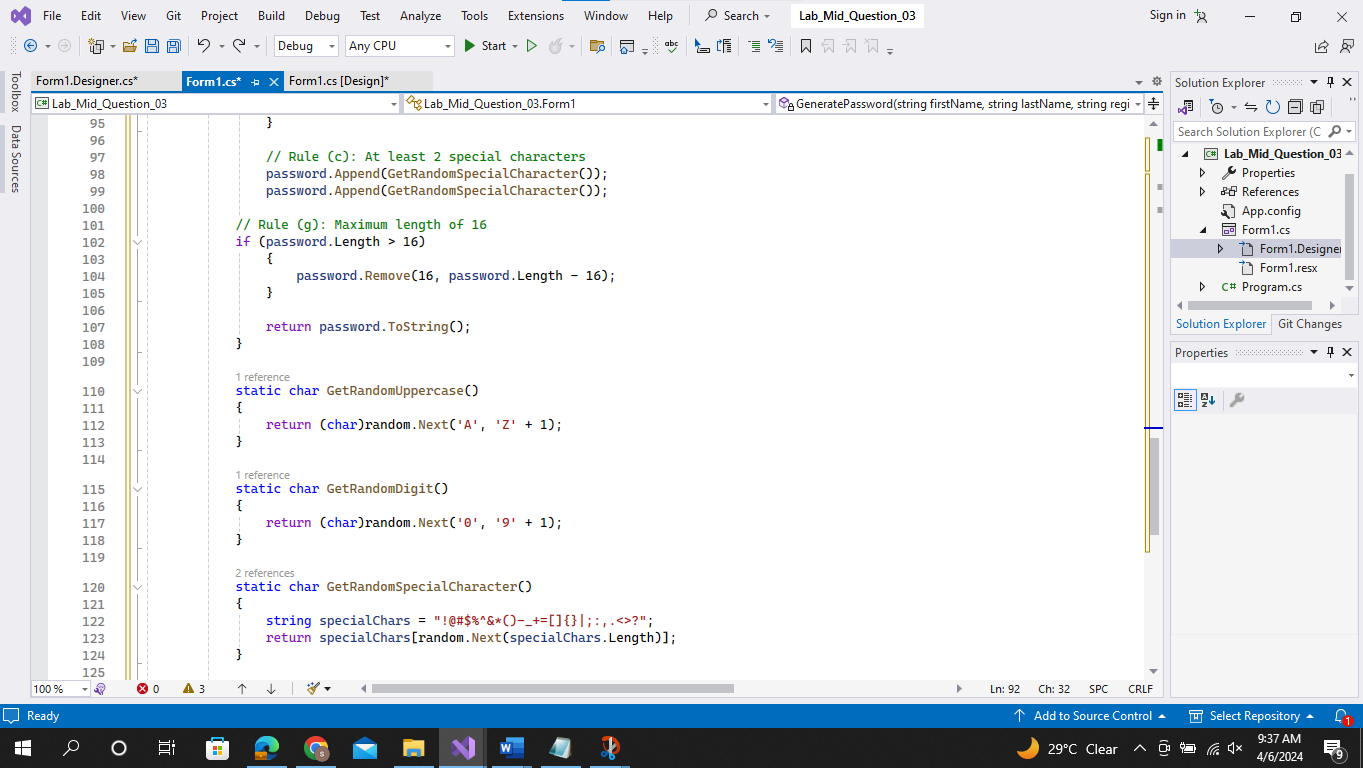
****

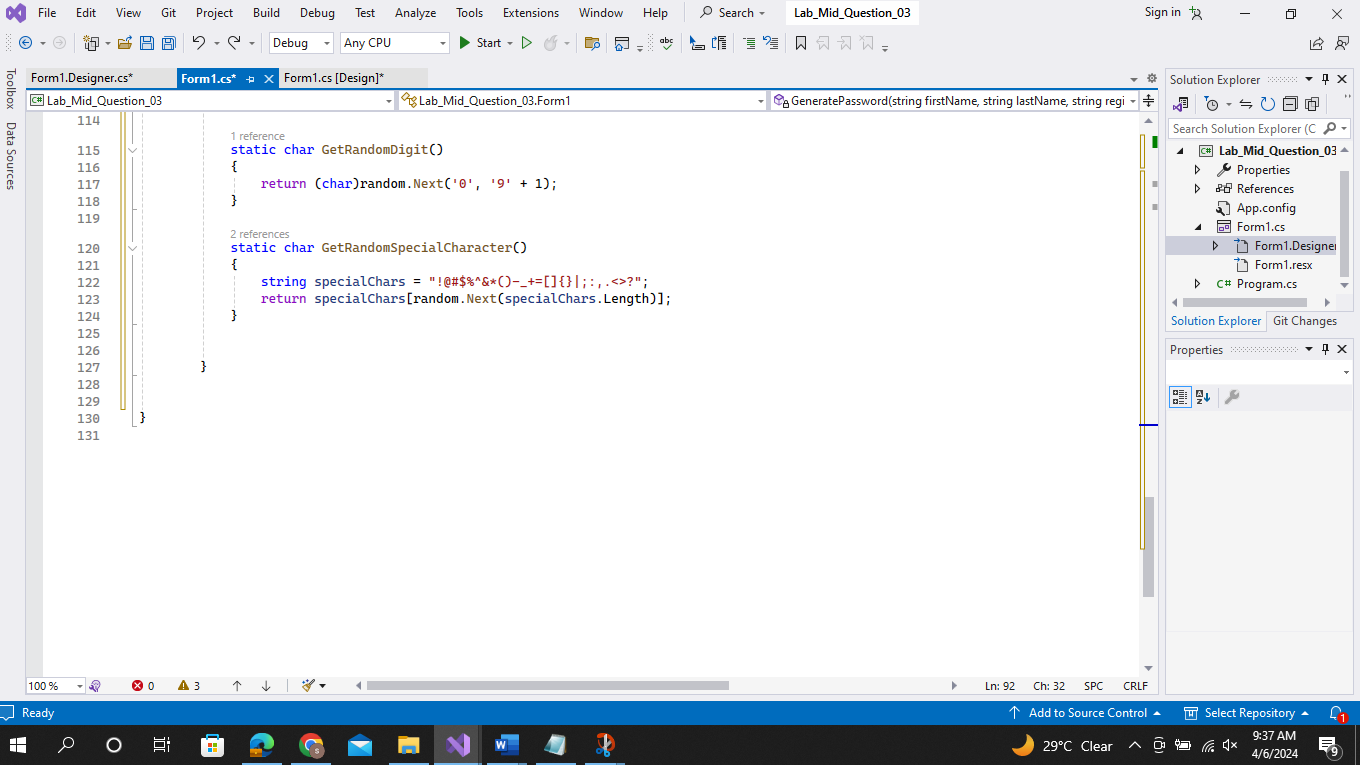
**Form.cs:**

****

****

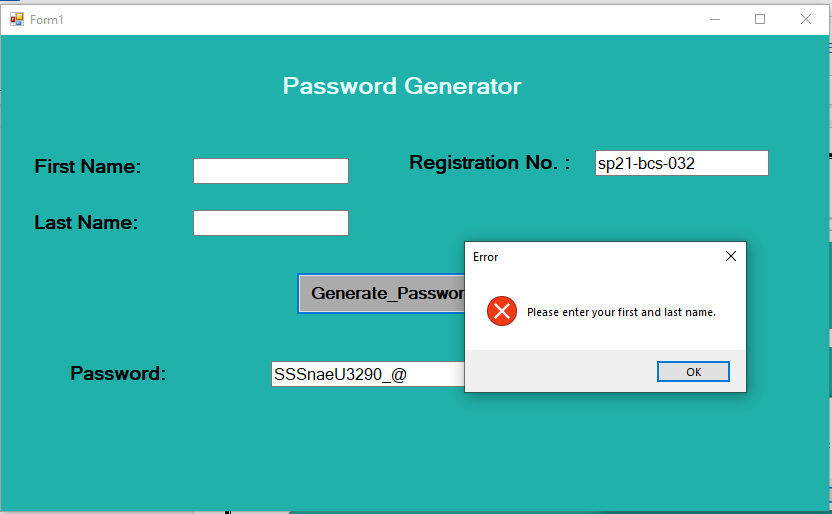
****

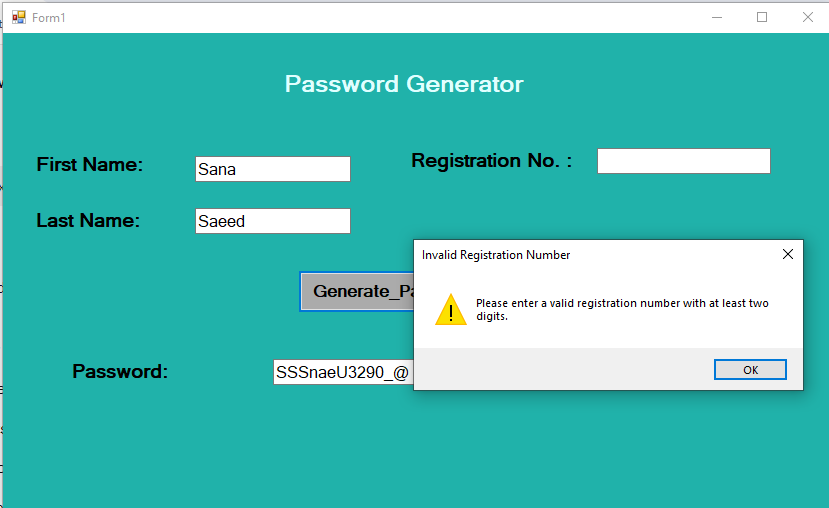
****

****

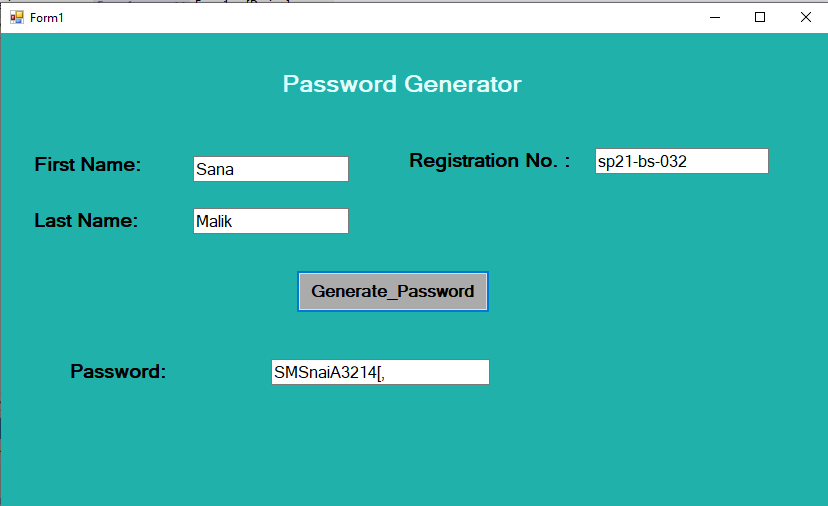
**Output:**

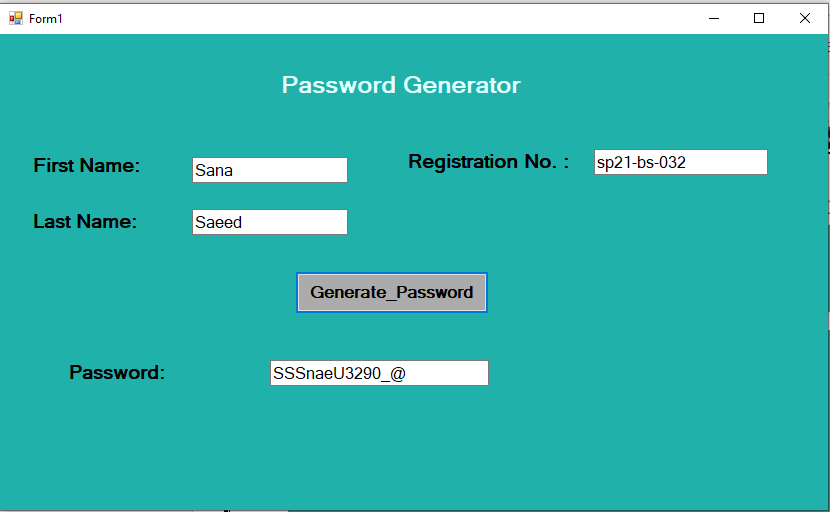
**Input Validations:**

****

****

**Generated Passwords:**

****

****