NAME: Amna Sajjad

CLASS: BCS-7A

REG NO: FA20-BCS-031

SUBJECT: COMPILER CONSTRUCTION

TEACHER: BILAL SAAB

Mid Lab

Date;26-oct-2023

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Question 1:**

**Describe functioning of regex C# library , give examples of patterns,seperators and anchors e.t.c.**

In C#, regular expressions are supported through the **`System.Text.RegularExpressions**` namespace, which provides classes and methods for working with regular expressions. Regular expressions are patterns used for matching and manipulating strings. They are powerful tools for tasks like searching, parsing, and validating textual data.

**1. Creating a Regex Object:**

To work with regular expressions in C#, you first need to create a `Regex` object, which represents the regular expression pattern. You can create it like this:

using System.Text.RegularExpressions;

string pattern = @"\d+"; // This is a simple pattern to match one or more digits

Regex regex = new Regex(pattern);

```

**2. Matching:**

The primary use of regular expressions is to search for patterns in strings. You can use the `Match` method to find the first occurrence of the pattern in a given input string: string input = "123abc456def";

Match match = regex.Match(input);

if (match.Success)

{

Console.WriteLine("Match found: " + match.Value); // Outputs: 123

}

```

**3. Matching Multiple Occurrences:**

To find all occurrences of the pattern in the input string, you can use the `Matches` method: string input = "123abc456def789";

MatchCollection matches = regex.Matches(input);

foreach (Match match in matches)

{

Console.WriteLine("Match found: " + match.Value);

}

**4. Anchors:**

Regular expressions support anchors to specify the position within the string where a match should occur. Common anchors are:

- `^` (caret) for the start of a line.

- `$` (dollar sign) for the end of a line.

- `\b` for word boundaries.

For example, to find all words that start with "cat" at the beginning of a line:

string pattern = @"^cat\w\*";

```

**5. Character Classes:**

Character classes allow you to match any character from a set of characters. For example, to match any vowel in a string:

string pattern = @"[aeiou]";

**6. Quantifiers:**

Quantifiers specify how many times a character or group should be matched. Some common quantifiers are:

- `\*` for zero or more occurrences.

- `+` for one or more occurrences.

- `?` for zero or one occurrence.

- `{n}` for exactly n occurrences.

- `{n,}` for at least n occurrences.

- `{n,m}` for between n and m occurrences.

For example, to match a date in the format MM/DD/YYYY:

string pattern = @"\d{2}/\d{2}/\d{4}";

Patterns and separators are fundamental concepts in regular expressions. Patterns define the structure you want to match in a string, and separators are characters or sequences that divide or delimit different parts of the string. Regular expressions use patterns to locate and manipulate text based on these separators. Here are examples of patterns and separators in regular expressions:

**Patterns:**

**1. Matching Digits**: To match one or more digits, you can use the pattern `\d+`. This pattern matches any sequence of one or more numeric digits.

Example:

string pattern = @"\d+";

**2. Matching Email Addresses:** A common pattern for matching email addresses is `[\w\.-]+@[\w\.-]+\.\w+`. This pattern matches the local part, the "@" symbol, the domain name, and the top-level domain (TLD).

**Example**:

string pattern = @"[\w\.-]+@[\w\.-]+\.\w+";

**3. Matching URLs:** To match URLs, you can use a pattern like `(https?|ftp)://[^\s/$.?#].[^\s]\*`. This pattern can match both HTTP and FTP URLs.

**Example:**

string pattern = @"(https?|ftp)://[^\s/$.?#].[^\s]\*";

**4. Matching Words**:To match words in a text, you can use the pattern `\w+`. This pattern matches one or more word characters (letters, digits, or underscores).

**Example:**

string pattern = @"\w+";

```

**Separators:**

**1. Commas:** The comma (`,`) is a common separator used in lists or CSV files to separate items.

Example:

string[] items = Regex.Split("apple,banana,cherry", ",");

**2. Whitespace**: Whitespace characters, such as spaces, tabs, and newlines, are often used as separators to split text into words or lines.

**Example:**

string[] words = Regex.Split("The quick brown fox", @"\s+");

**3.Colons**: Colons (`:`) are often used to separate key-value pairs in data structures.

**Example:**

string[] keyValue = Regex.Split("name:John;age:30;city:New York", ";");

4. \*\*Periods\*\*: Periods (`.`) are used as separators in file extensions.

**Example:**

string[] extensions = Regex.Split("document.pdf,image.jpg,report.docx", ",");

**5. Slashes and Backslashes:** Forward slashes (`/`) and backslashes (`\`) are used as separators in file paths and URLs.

**Example:**

string[] filePathParts = Regex.Split("C:\\Folder\\File.txt", @"[/\\]");

Regular expressions are powerful tools for working with patterns and separators in text. You can use patterns to specify what you want to match, and separators to split or extract specific parts of a string based on those patterns.

**Question 2**

**For the given grammar create LL(1) or recursive descent parser**

**List -> Item Rest**

**Rest -> , Item Rest | ε**

**Item -> id | num | string**

**Answer:**

First, let's create the First and Follow sets for the non-terminals:

* First(List) = {id, num, string}
* First(Rest) = {,, ε}
* First(Item) = {id, num, string}
* Follow(List) = {$}
* Follow(Rest) = {$}
* Follow(Item) = {,, $}

**Parsing table of LL1:**

Parsing table of LL1:

**Parsing Table:**

------------------------------------------------------------

| | id | num | string | , | $ |

------------------------------------------------------------

| List | List -> Item Rest | List -> Item Rest | List -> Item Rest | | |

------------------------------------------------------------

| Rest | ε | ε | ε | , Item Rest | ε |

------------------------------------------------------------

| Item | Item -> id | Item -> num | Item -> string | | |

------------------------------------------------------------

Code:

using System;

class LL1Parser

{

private string[] input;

private int index;

public LL1Parser(string[] input)

{

this.input = input;

this.index = 0;

}

public void ParseList()

{

ParseItem();

ParseRest();

if (index == input.Length) // Check if we have reached the end of the input.

Console.WriteLine("Parsing completed successfully.");

else

Console.WriteLine("Parsing failed.");

}

private void ParseRest()

{

if (index < input.Length && input[index] == ",")

{

Match(",");

ParseItem();

ParseRest();

}

// For Rest -> ε (empty)

}

private void ParseItem()

{

if (index < input.Length && (input[index] == "id" || input[index] == "num" || input[index] == "string"))

{

index++;

}

else

{

Console.WriteLine("Parsing failed.");

}

}

private void Match(string expectedToken)

{

if (index < input.Length && input[index] == expectedToken)

{

index++;

}

else

{

Console.WriteLine("Parsing failed.");

}

}

public static void Main(string[] args)

{

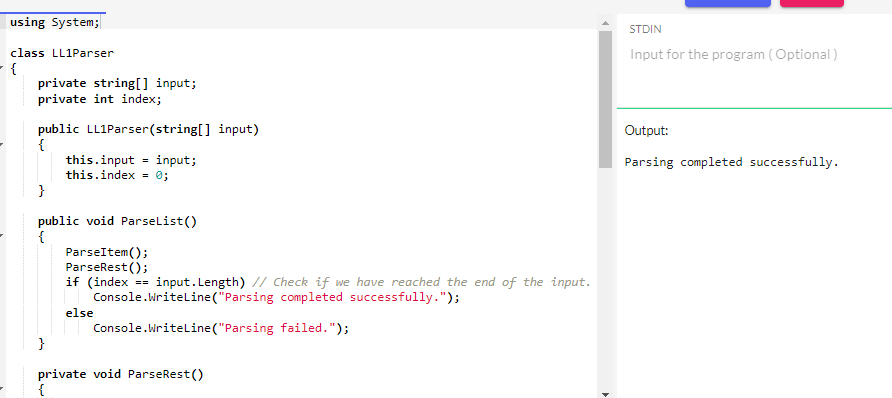
string[] input = { "id", ",", "num", ",", "string" };

LL1Parser parser = new LL1Parser(input);

parser.ParseList();

}

}



**Question3:**

Create a password generator which has the following rules

1. maximum length of 20

2. Atleast 2 special characters

3. Atleast 4 numbers

4. Intials of your first and last name (e.g. Babar Khan has intials B and K) in upper case

5. 2 of the numbers should be your last two digits of your registration number (e.g. if registration number is fa20-bcs-012)

then two of the numbers should be 1 and 2

**Answer:**

**Code:**

using System;

using System.Text;

using System.Linq;

class PasswordGenerator

{

static void Main(string[] args)

{

// Replace these values with your actual information

string firstName = "Amna";

string lastName = "Sajjad";

string registrationNumber = "fa20-bcs-031";

// Rule 1: Maximum length of 20

int maxLength = 20;

// Rule 2: At least 2 special characters

string specialCharacters = "!@#$%^&\*()\_-+=<>?";

// Rule 4: Initials of your first and last name

string initials = (firstName[0].ToString() + lastName[0].ToString()).ToUpper();

// Rule 5: Last two digits of your registration number

string lastTwoDigits = new string(registrationNumber.Where(char.IsDigit).ToArray());

lastTwoDigits = lastTwoDigits.Substring(Math.Max(0, lastTwoDigits.Length - 2));

// Rule 3: At least 4 numbers (2 of them from your registration number)

int remainingNumbers = 4 - lastTwoDigits.Length;

string numbers = lastTwoDigits + "12" + new string('0', remainingNumbers);

// Create a StringBuilder to build the password

StringBuilder password = new StringBuilder();

// Add the initials

password.Append(initials);

// Add 2 special characters

Random random = new Random();

for (int i = 0; i < 2; i++)

{

password.Append(specialCharacters[random.Next(specialCharacters.Length)]);

}

// Shuffle the numbers to ensure randomness

numbers = new string(numbers.OrderBy(c => random.Next()).ToArray());

// Add the numbers

password.Append(numbers);

// Generate the remaining part of the password with random characters

int remainingLength = maxLength - password.Length;

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

{

password.Append(specialCharacters[random.Next(specialCharacters.Length)]);

}

// Shuffle the password to make it more secure

string shuffledPassword = ShufflePassword(password.ToString());

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

}

// Function to shuffle the characters in the password

static string ShufflePassword(string password)

{

char[] passwordArray = password.ToCharArray();

Random random = new Random();

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

{

int j = random.Next(i, passwordArray.Length);

char temp = passwordArray[i];

passwordArray[i] = passwordArray[j];

passwordArray[j] = temp;

}

return new string(passwordArray);

}

}

