**COMSATS UNIVERSITY ISLAMABAD ATTOCK CAMPUS**

****

**LAB MID TERM**

**COMPILER CONSTRUCTION**

**Name: SAMRA**

**RegNo: SP21-BCS-030**

**Submitted To: Sir, Bilal Haider Bukhari**

**Date: 7th April, 2024**

**QUESTION NO 1:**

**Briefly describe the regex library of C#:**

In C#, the regex library provides a powerful tool for working with regular expressions, allowing developers to search, match, and manipulate text based on patterns defined by regular expressions. Here's a brief overview of the regex library in C#:

1. **Namespace:**

The regex library is part of the `System.Text.RegularExpressions` namespace.

1. **Classes:**

The main classes in the regex library are `Regex` and `Match`.

`Regex`: Represents a compiled regular expression pattern.

`Match`: Represents the results of a single match operation.

**3. Functionality:**

Pattern Matching: You can use regex patterns to search for specific patterns within strings.

Pattern Replacement: You can replace matches found in strings with other patterns.

Pattern Validation: You can use regular expressions to validate strings against certain patterns.

Grouping: You can use capturing groups to extract specific parts of a matched pattern.

Options: Regex options such as case-insensitivity and single-line mode can be specified to modify matching behavior.

Anchors and Quantifiers: Regex supports anchors like `^` (start of string) and `$` (end of string) and quantifiers like `\*`, `+`, `?`, and `{}` for specifying repetition.

**4. Methods:**

- `Regex.Match()`: Searches for the first occurrence of a pattern in a string.

- `Regex.Matches()`: Searches for all occurrences of a pattern in a string.

- `Regex.Replace()`: Replaces all occurrences of a pattern in a string with a specified replacement.

`Regex.IsMatch()`: Determines whether a string matches a given pattern.

`Match.Groups`: Provides access to captured groups within a match.

**5. Pattern Syntax:**

C# regex patterns follow the standard syntax for regular expressions, including metacharacters, character classes, quantifiers, anchors, and grouping constructs.

**6. Options:**

`RegexOptions.IgnoreCase`: Enables case-insensitive matching.

`RegexOptions.Multiline`: Treats the string as consisting of multiple lines.

`RegexOptions.Singleline`: Treats the input string as a single line.

`RegexOptions.IgnorePatternWhitespace`: Allows whitespace and comments within the pattern.

Overall, the regex library in C# provides comprehensive support for working with regular expressions, making it a valuable tool for tasks such as text searching, validation, and manipulation.

**QUESTION NO 2:**

**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

**NOTE:**

The above grammar is left recursive first convert it into right recursive then make LL1 parser.

**Right-recursive Grammar:**

S -> X$

X -> Y X'

X' -> % Y X' | ε

Y -> Z Y'

Y' -> & Z Y' | ε

Z -> k X k Z' | g Z'

Z' -> k Z' | ε

**CODE:**

#include <iostream>

#include <string>

using namespace std;

class Parser {

private:

string input\_string;

size\_t index;

char current\_token;

public:

Parser(const string& input) : input\_string(input), index(0), current\_token(input[0]) {}

bool match(char expected\_token) {

if (current\_token == expected\_token) {

index++;

if (index < input\_string.length()) {

current\_token = input\_string[index];

}

return true;

}

return false;

}

bool parse\_S() {

if (parse\_X() && match('$')) {

return true;

}

return false;

}

bool parse\_X() {

if (parse\_Y() && parse\_X\_prime()) {

return true;

}

return false;

}

bool parse\_X\_prime() {

if (match('%')) {

if (parse\_Y() && parse\_X\_prime()) {

return true;

} }

return true;

}

bool parse\_Y() {

if (parse\_Z() && parse\_Y\_prime()) {

return true;

}

return false;

}

bool parse\_Y\_prime() {

if (match('&')) {

if (parse\_Z() && parse\_Y\_prime()) {

return true;

} }

return true;

}

bool parse\_Z() {

if (match('k') && parse\_X() && match('k')) {

return true;

}

else if (match('g')) {

return true;

}

return false;

}

bool parse() {

return parse\_S();

}};

int main() {

string input;

cout << "Enter a string to parse: ";

cin >> input;

Parser parser(input);

if (parser.parse()) {

cout << "String Accepted because String is in the language" << endl;

}

else {

cout << "Rejected , because String is not in the language" << endl;

}

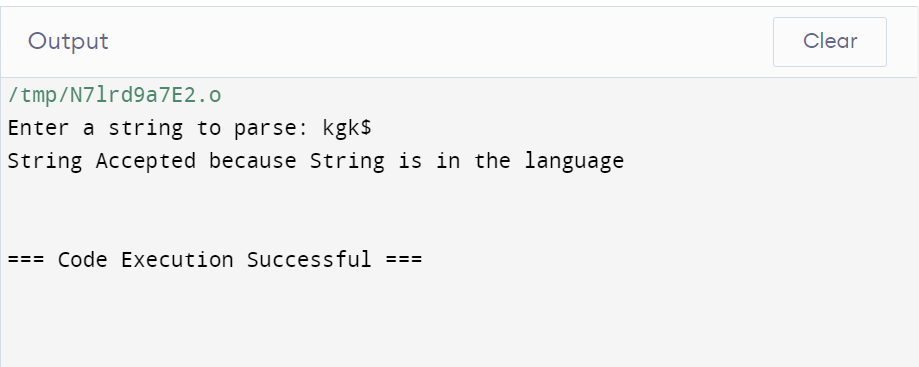
return 0;

}

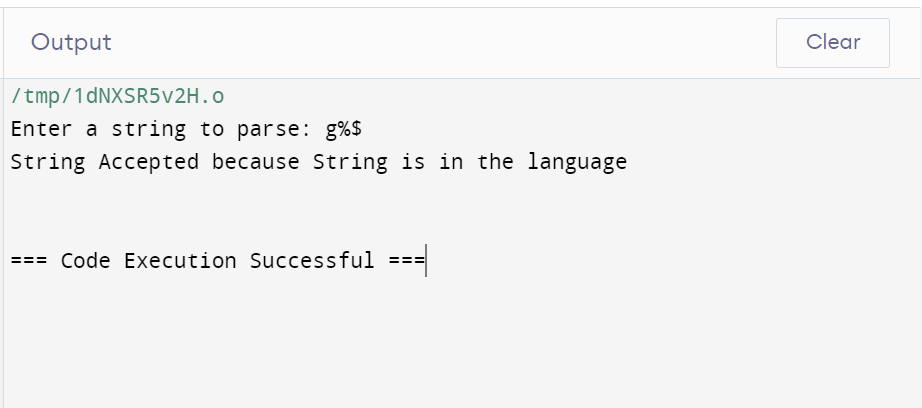
**INPUT & OUTPUT:**

**Accepted String By Grammar.**

**kgk$**

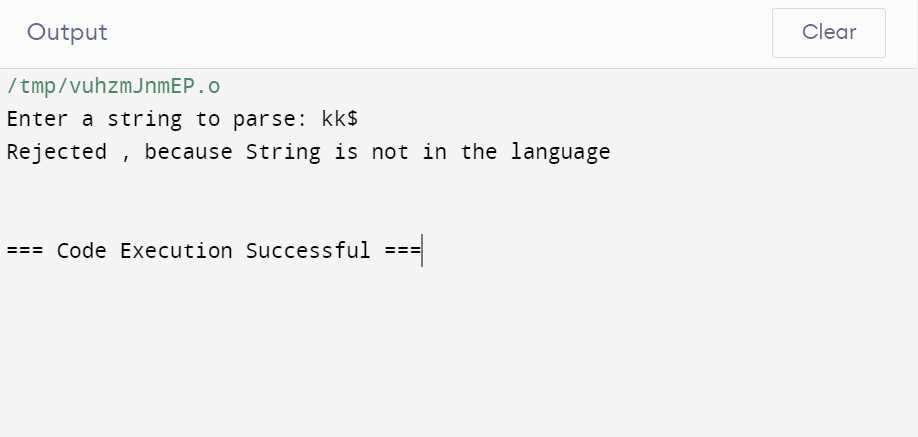
****

**g%$**

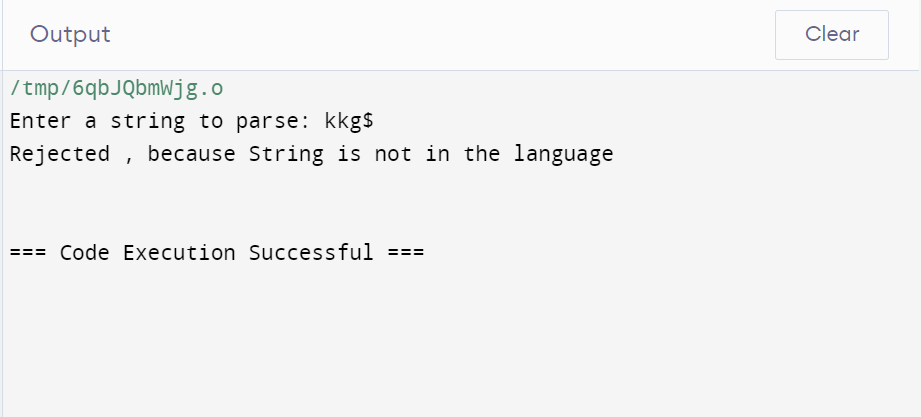
****

**Rejected:**

kk$

****

**kkg$**

****

**QUESTION NO 3:**

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

**CODE:**

**C++ :**

#include <iostream>

#include <string>

#include <vector>

#include <random>

#include <algorithm>

char generateRandomUppercase() {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis('A', 'Z');

return static\_cast<char>(dis(gen));

}

char generateRandomDigit() {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(0, 9);

return static\_cast<char>(dis(gen) + '0');

}

// Function to generate a random special character

char generateRandomSpecialChar() {

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

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(0, specialChars.size() - 1);

return specialChars[dis(gen)];

}

std::string generatePassword(const std::string& first\_name, const std::string& last\_name, const std::vector<int>& reg\_numbers) {

std::string password;

// At least one uppercase alphabet

password += generateRandomUppercase();

// At least 4 numbers, including two registration numbers

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

password += std::to\_string(reg\_numbers[i]);

}

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

password += generateRandomDigit();

}

// At least 2 special characters

password += generateRandomSpecialChar();

password += generateRandomSpecialChar();

// Must contain initials of first and last name

password += std::toupper(first\_name[0]);

password += std::toupper(last\_name[0]);

// Must contain all odd letters of the first name

for (int i = 0; i < first\_name.size(); ++i) {

if (i % 2 == 0) {

password += first\_name[i];

}

}

// Must contain all even letters of the last name

for (int i = 0; i < last\_name.size(); ++i) {

if (i % 2 != 0) {

password += last\_name[i];

}

}

// Shuffle the password

std::random\_device rd;

std::mt19937 g(rd());

std::shuffle(password.begin(), password.end(), g);

// Truncate if necessary

if (password.size() > 16) {

password = password.substr(0, 16);

}

return password;

}

int main() {

std::string first\_name = "John";

std::string last\_name = "Doe";

std::vector<int> reg\_numbers = {1234, 5678}; // Example registration numbers

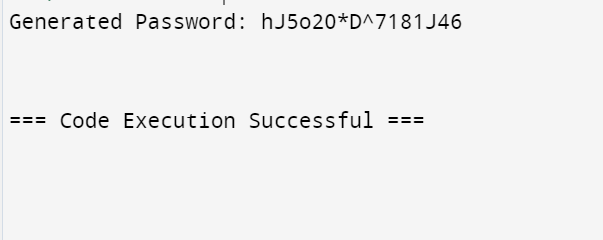
std::string password = generatePassword(first\_name, last\_name, reg\_numbers);

std::cout << "Generated Password: " << password << std::endl;

return 0;

}

**OUTPUT:**

****