|  |  |
| --- | --- |
|  | **COMSATS University Islamabad, Attock Campus**  **Lab Terminal Examinations (Spring 2024)** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Department of: | | **Computer Science** | | | | |  |
|  | |  | |  | | | | |  |
| Class/Program: | | **BS(CS)-7th** | | Date: | **31/05/ 2024 (1:30 - 4:30)** | | | | |
| Subject: | **Compiler construction Lab EXAM** | | | Instructor: | | **Bilal Haider** | | | |
| Total Time Allowed: | | | **3Hrs** | Maximum Marks: | | | | **50** | |
| Student Name: | | Sadia Bibi | | Registration #: | | | SP21-BCS-027 | | |
|  | |  | |  | | |  | | |

**Question 1**

**Write an introduction of your compiler construction project**

**Project Introduction: Custom Compiler Construction**

This project is dedicated to developing a basic compiler for a custom-designed programming language using C++. The compiler's primary objective is to translate source code written in this custom language into JavaScript code, enabling its execution within a JavaScript runtime environment.

**Custom Programming Language**

The custom programming language designed for this project includes basic constructs typical of many programming languages:

• Function Definitions: Declared with a def keyword and containing a name, parameters, and a body.

• Basic Arithmetic Operations: Supports operations such as addition and multiplication.

• Function Calls: Allows calling functions with specified arguments.

Key Components of the Compiler

1. **Tokenizer (Lexical Analyzer):**

Role: The tokenizer scans the source code to identify and extract tokens, which are the smallest meaningful units of the code. Tokens include keywords (like def), identifiers (like variable names), numeric literals, and operators.

Functionality: By using regular expressions, the tokenizer matches patterns in the source code and categorizes them into different token types. This process simplifies the subsequent parsing stage by providing a structured sequence of tokens.

2. **Parser:**

Role: The parser takes the sequence of tokens generated by the tokenizer and constructs an abstract syntax tree (AST). The AST is a hierarchical representation of the program's syntax and structure.

Functionality: The parser uses grammar rules specific to the custom language to recognize constructs such as function definitions, expressions, and statements. It organizes these constructs into nodes of the AST, reflecting the nested and sequential relationships inherent in the source code.

3. **Abstract Syntax Tree (AST):**

Role: The AST serves as an intermediate representation of the source code, capturing its syntactic structure in a tree format. Each node in the tree corresponds to a specific language construct.

Node Types: The AST includes various node types, each representing different elements of the language:

DefNode: Represents function definitions.

IntNode: Represents integer literals.

CallNode: Represents function calls.

VarNode: Represents variable references.

OpNode: Represents arithmetic operations.

Structure: The AST provides a clear and organized view of the code's structure, facilitating the translation process in the code generation phase.

4. **Code Generator:**

Role: The code generator traverses the AST and produces equivalent JavaScript code. This phase ensures that the functionality of the original custom language code is preserved in the generated JavaScript.

Functionality: The code generator recursively processes each node of the AST:

DefNode: Generates JavaScript function definitions.

IntNode: Generates JavaScript integer literals.

CallNode: Generates JavaScript function calls.

VarNode: Generates JavaScript variable references.

OpNode: Generates JavaScript expressions for arithmetic operations.

Output: The final output is a string containing JavaScript code that mirrors the logic and structure of the original custom language program.

**Question 2**

**Give a sample input and output for your compiler construction project**

**Project CODE:**

#include <iostream>

#include <regex>

#include <fstream>

#include <sstream>

#include <deque>

#include <stdexcept>

using namespace std;

void trim(string &str);

struct Token{

enum Type {DEF, END, IDENTIFIER, INTEGER, OPAREN, CPAREN, COMMA, OP};

Type type;

string value;

Token(Type type, string value) {

this->type = type;

this->value = value;

}

friend ostream& operator<<(ostream& os, const Token &t) {

return os << "type=" << t.type << ",value=" << t.value << endl;

}

};

class Tokenizer {

public:

static const int N = 8;

const string re[N] = {"^(\\bdef\\b)", "^(\\bend\\b)", "^(\\b[a-zA-Z]+\\b)", "^(\\b[0-9]+\\b)", "^\\(", "^\\)", "\\,", "[\\+, \\-, \\/, \\\*]"};

string code;

Tokenizer(string fname) {

string snippet;

ifstream file(fname);

if (!file) {

throw runtime\_error("File not found: " + fname);

}

std::stringstream sstr;

sstr << file.rdbuf();

this->code = sstr.str();

this->code.erase(remove(this->code.begin(), this->code.end(), '\n'), this->code.end());

}

deque<Token> tokenize() {

deque<Token> tokens;

while (!this->code.empty()) {

Token token = tokenize\_one();

tokens.push\_back(token);

}

return tokens;

}

Token tokenize\_one() {

smatch sm;

string result;

for (int t = 0; t < N; t++) {

if (regex\_search(this->code, sm, regex(re[t]))) {

result = sm.str(0);

this->code.erase(this->code.begin(), this->code.begin() + result.length());

trim(this->code);

return Token(static\_cast<Token::Type>(t), result);

}

}

throw runtime\_error("Failed to tokenize: " + this->code);

}

};

class Node {

public:

enum Type {DEFNODE, INTNODE, CALLNODE, VARNODE, OPNODE};

virtual ~Node() {}

virtual void print() = 0;

virtual Type type() const = 0;

};

class ExprNode : public Node {

public:

virtual void print() = 0;

virtual Type type() const { return DEFNODE; }

};

class DefNode : public Node {

public:

string name;

deque<string> args;

ExprNode\* body;

DefNode(string n, deque<string> a, ExprNode\* b) : name(n), args(a), body(b) {}

Type type() const { return DEFNODE; }

void print() {

cout << "name=" << this->name << ", args=";

for (const auto& arg : this->args) {

cout << arg << ", ";

}

cout << ", body=";

this->body->print();

cout << endl;

}

};

class IntNode : public ExprNode {

public:

string value;

IntNode(string v) : value(v) {}

Type type() const { return INTNODE; }

void print() {

cout << "[(IntNode)value=" << this->value << "]";

}

};

class CallNode : public ExprNode {

public:

string name;

deque<ExprNode\*> arg\_exprs;

CallNode(string n, deque<ExprNode\*> a) : name(n), arg\_exprs(a) { }

Type type() const { return CALLNODE; }

void print() {

cout << "[(CallNode)name=" << name << ", arg\_exprs=";

for (const auto& expr : arg\_exprs) {

expr->print();

}

cout << "]";

}

};

class VarNode : public ExprNode {

public:

string name;

VarNode(string n) : name(n) { }

Type type() const { return VARNODE; }

void print() {

cout << "[(VarNode)name=" << name << "]";

}

};

class OpNode : public ExprNode {

public:

string op;

ExprNode\* arg\_expr1;

ExprNode\* arg\_expr2;

OpNode(string o, ExprNode\* a1, ExprNode\* a2) : op(o), arg\_expr1(a1), arg\_expr2(a2) { }

Type type() const { return OPNODE; }

void print() {

cout << "[(OpNode)op=" << op << ", arg\_exprs=(";

arg\_expr1->print();

cout << ", ";

arg\_expr2->print();

cout << ")]";

}

};

class Parser {

public:

deque<Token> tokens;

Parser(deque<Token> tokens) : tokens(tokens) {}

DefNode\* parse\_def() {

consume(Token::DEF);

string name = consume(Token::IDENTIFIER);

deque<string> arg\_names = parse\_arg\_names();

ExprNode\* body = parse\_expr();

return new DefNode(name, arg\_names, body);

}

deque<string> parse\_arg\_names() {

consume(Token::OPAREN);

deque<string> arg\_names;

if (peek(Token::IDENTIFIER)) {

arg\_names.push\_back(consume(Token::IDENTIFIER));

while (peek(Token::COMMA)) {

consume(Token::COMMA);

arg\_names.push\_back(consume(Token::IDENTIFIER));

}

}

consume(Token::CPAREN);

return arg\_names;

}

ExprNode\* parse\_expr() {

if (peek(Token::INTEGER)) {

return parse\_int();

} else if (peek(Token::IDENTIFIER) && peek(Token::OPAREN, 1)) {

return parse\_call();

} else if (peek(Token::OP)) {

return parse\_op();

} else {

return parse\_var\_ref();

}

}

IntNode\* parse\_int() {

string i = consume(Token::INTEGER);

return new IntNode(i);

}

CallNode\* parse\_call() {

string name = consume(Token::IDENTIFIER);

deque<ExprNode\*> arg\_exprs = parse\_arg\_exprs();

return new CallNode(name, arg\_exprs);

}

deque<ExprNode\*> parse\_arg\_exprs() {

consume(Token::OPAREN);

deque<ExprNode\*> arg\_exprs;

if (!peek(Token::CPAREN)) {

arg\_exprs.push\_back(parse\_expr());

while (peek(Token::COMMA)) {

consume(Token::COMMA);

arg\_exprs.push\_back(parse\_expr());

}

}

consume(Token::CPAREN);

return arg\_exprs;

}

VarNode\* parse\_var\_ref() {

string var = consume(Token::IDENTIFIER);

return new VarNode(var);

}

OpNode\* parse\_op() {

string op = consume(Token::OP);

ExprNode\* arg\_expr1 = parse\_expr();

ExprNode\* arg\_expr2 = parse\_expr();

return new OpNode(op, arg\_expr1, arg\_expr2);

}

DefNode\* parse() {

return parse\_def();

}

string consume(Token::Type type) {

if (tokens.empty()) {

throw runtime\_error("Unexpected end of input");

}

Token token = this->tokens.front();

this->tokens.pop\_front();

if (token.type == type) {

return token.value;

} else {

char buf[100];

sprintf(buf, "Could not parse given type %d, expected %d", token.type, type);

throw runtime\_error(buf);

}

}

bool peek(Token::Type type, int offset = 0) {

if (offset >= tokens.size()) {

return false;

}

return this->tokens.at(offset).type == type;

}

};

class Generator {

public:

Node\* tree;

Generator(Node\* t) : tree(t) {}

string generate() {

return generate\_helper(this->tree);

}

string generate\_helper(Node\* node) {

switch (node->type()) {

case Node::DEFNODE: {

DefNode\* defnode = static\_cast<DefNode\*>(node);

string args;

if (!defnode->args.empty()) {

args = defnode->args.front();

for (size\_t i = 1; i < defnode->args.size(); ++i) {

args += ", " + defnode->args[i];

}

}

return "function " + defnode->name + "(" + args + ") {return " + generate\_helper(defnode->body) + "};";

}

case Node::INTNODE: {

IntNode\* intnode = static\_cast<IntNode\*>(node);

return intnode->value;

}

case Node::CALLNODE: {

CallNode\* callnode = static\_cast<CallNode\*>(node);

string arg\_exprs;

if (!callnode->arg\_exprs.empty()) {

arg\_exprs = generate\_helper(callnode->arg\_exprs.front());

for (size\_t i = 1; i < callnode->arg\_exprs.size(); ++i) {

arg\_exprs += ", " + generate\_helper(callnode->arg\_exprs[i]);

}

}

return callnode->name + "(" + arg\_exprs + ")";

}

case Node::VARNODE: {

VarNode\* varnode = static\_cast<VarNode\*>(node);

return varnode->name;

}

case Node::OPNODE: {

OpNode\* opnode = static\_cast<OpNode\*>(node);

return "(" + generate\_helper(opnode->arg\_expr1) + opnode->op + generate\_helper(opnode->arg\_expr2) + ")";

}

default:

char buf[100];

sprintf(buf, "Could not generate for given class type %d", node->type());

throw runtime\_error(buf);

}

}

};

void trim(string& str) {

auto it = str.begin();

while (it != str.end() && \*it == ' ') {

++it;

}

str.erase(str.begin(), it);

}

int main() {

try {

Tokenizer t = Tokenizer("test.lang");

deque<Token> tokens = t.tokenize();

Parser parser = Parser(tokens);

DefNode\* tree = parser.parse();

// tree->print();

Generator generator = Generator(tree);

string generated = generator.generate();

string TEST = "console.log(f(1,2));";

cout << generated << "\n" << TEST << endl;

} catch (const std::exception& e) {

cerr << "Error: " << e.what() << endl;

return 1;

}

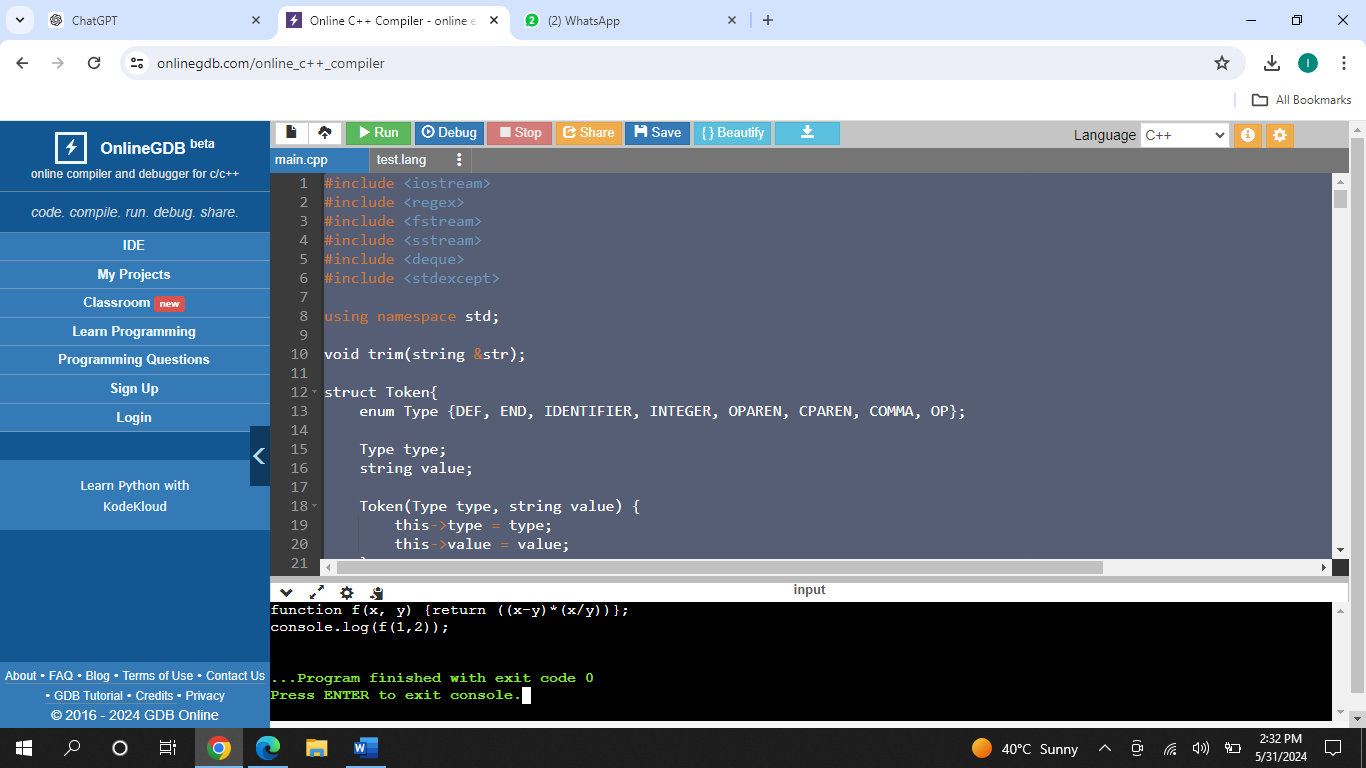
return 0;

}

**TEST INPUT:**

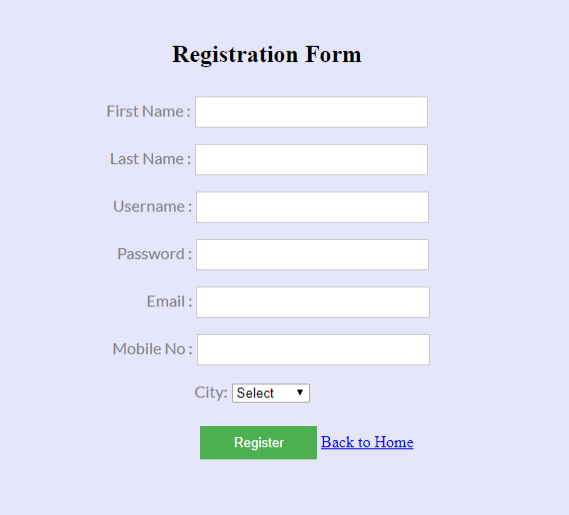
def f(x,y) \* - x y / x y end

**OUTPUT:**



**Question 3**

**Create and implement RE and DFAs for the form below**



You must use Regex to validate data.

**Code:**

<!DOCTYPE html>

<html>

<head>

<title>Registration Form</title>

<style>

body {

background-color: #E6E6FA;

font-family: Arial, sans-serif;

}

.container {

width: 300px;

margin: 0 auto;

padding-top: 50px;

}

form {

background: #FFF;

padding: 20px;

border-radius: 10px;

box-shadow: 0px 0px 10px 0px #000;

}

label, input, select {

display: block;

width: 100%;

margin-bottom: 10px;

}

button {

background-color: green;

color: white;

padding: 10px;

border: none;

border-radius: 5px;

cursor: pointer;

}

.error {

color: red;

font-size: 0.9em;

margin-top: -10px;

margin-bottom: 10px;

}

</style>

</head>

<body>

<div class="container">

<form id="registrationForm">

<h2>Registration Form</h2>

<label>First Name: <input type="text" id="firstName" required></label>

<span class="error" id="firstNameError"></span>

<label>Last Name: <input type="text" id="lastName" required></label>

<span class="error" id="lastNameError"></span>

<label>Username: <input type="text" id="username" required></label>

<span class="error" id="usernameError"></span>

<label>Password: <input type="password" id="password" required></label>

<span class="error" id="passwordError"></span>

<label>Email: <input type="email" id="email" required></label>

<span class="error" id="emailError"></span>

<label>Mobile No: <input type="text" id="mobile" required></label>

<span class="error" id="mobileError"></span>

<label>City:

<select id="city" required>

<option value="Select">Select</option>

<option value="New York">New York</option>

<option value="Los Angeles">Los Angeles</option>

<option value="Chicago">Chicago</option>

</select>

</label>

<span class="error" id="cityError"></span>

<button type="submit">Register</button>

</form>

<a href="index.html">Back to Home</a>

</div>

<script>

const form = document.getElementById('registrationForm');

form.addEventListener('submit', function(event) {

// Regular Expressions

const nameRegex = /^[A-Za-z]+$/;

const usernameRegex = /^[A-Za-z0-9\_]{3,16}$/;

const passwordRegex = /^(?=.\*[A-Z])(?=.\*[a-z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/;

const emailRegex = /^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

const mobileRegex = /^\d{10}$/;

const cityRegex = /^(Select|New York|Los Angeles|Chicago)$/;

// Input Values

const firstName = document.getElementById('firstName').value;

const lastName = document.getElementById('lastName').value;

const username = document.getElementById('username').value;

const password = document.getElementById('password').value;

const email = document.getElementById('email').value;

const mobile = document.getElementById('mobile').value;

const city = document.getElementById('city').value;

// Error Messages

const firstNameError = document.getElementById('firstNameError');

const lastNameError = document.getElementById('lastNameError');

const usernameError = document.getElementById('usernameError');

const passwordError = document.getElementById('passwordError');

const emailError = document.getElementById('emailError');

const mobileError = document.getElementById('mobileError');

const cityError = document.getElementById('cityError');

// Reset Error Messages

firstNameError.textContent = '';

lastNameError.textContent = '';

usernameError.textContent = '';

passwordError.textContent = '';

emailError.textContent = '';

mobileError.textContent = '';

cityError.textContent = '';

// Validation

let isValid = true;

if (!nameRegex.test(firstName)) {

firstNameError.textContent = 'Invalid First Name';

isValid = false;

}

if (!nameRegex.test(lastName)) {

lastNameError.textContent = 'Invalid Last Name';

isValid = false;

}

if (!usernameRegex.test(username)) {

usernameError.textContent = 'Invalid Username';

isValid = false;

}

if (!passwordRegex.test(password)) {

passwordError.textContent = 'Invalid Password';

isValid = false;

}

if (!emailRegex.test(email)) {

emailError.textContent = 'Invalid Email';

isValid = false;

}

if (!mobileRegex.test(mobile)) {

mobileError.textContent = 'Invalid Mobile Number';

isValid = false;

}

if (!cityRegex.test(city)) {

cityError.textContent = 'Invalid City';

isValid = false;

}

if (!isValid) {

event.preventDefault();

} else {

alert('Registration Successful!');

}

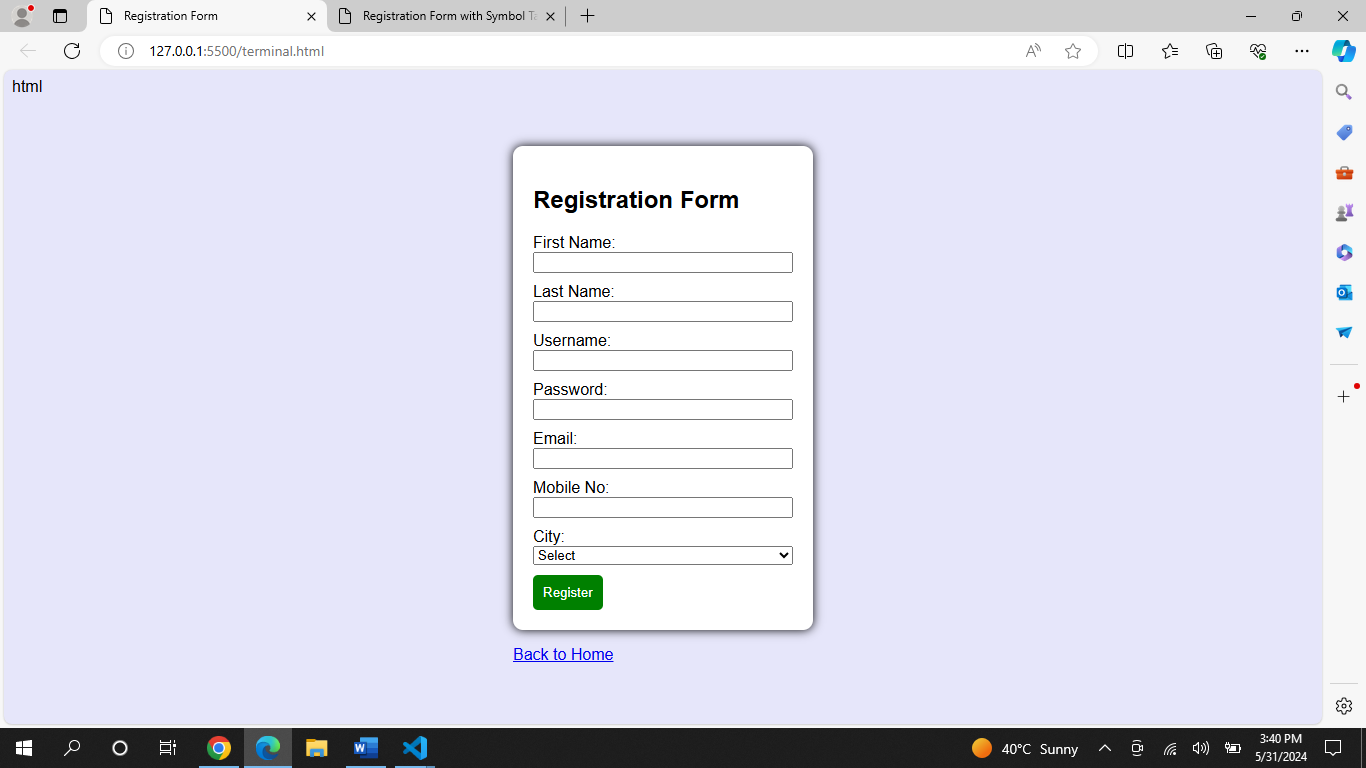
});

</script>

</body>

</html>

**Output :**



**Question 4:**

**Write a program which generates symbol table for the code you submitted in question 3.**

**Code:**

html

<!DOCTYPE html>

<html>

<head>

<title>Registration Form with Symbol Table</title>

<style>

body {

background-color: #E6E6FA;

font-family: Arial, sans-serif;

}

.container {

width: 300px;

margin: 0 auto;

padding-top: 50px;

}

form {

background: #FFF;

padding: 20px;

border-radius: 10px;

box-shadow: 0px 0px 10px 0px #000;

}

label, input, select {

display: block;

width: 100%;

margin-bottom: 10px;

}

button {

background-color: green;

color: white;

padding: 10px;

border: none;

border-radius: 5px;

cursor: pointer;

}

.error {

color: red;

font-size: 0.9em;

margin-top: -10px;

margin-bottom: 10px;

}

.symbol-table {

margin-top: 20px;

background: #FFF;

padding: 10px;

border-radius: 10px;

box-shadow: 0px 0px 10px 0px #000;

}

.symbol-table table {

width: 100%;

border-collapse: collapse;

}

.symbol-table th, .symbol-table td {

border: 1px solid #000;

padding: 5px;

text-align: left;

}

</style>

</head>

<body>

<div class="container">

<form id="registrationForm">

<h2>Registration Form</h2>

<label>First Name: <input type="text" id="firstName" required></label>

<span class="error" id="firstNameError"></span>

<label>Last Name: <input type="text" id="lastName" required></label>

<span class="error" id="lastNameError"></span>

<label>Username: <input type="text" id="username" required></label>

<span class="error" id="usernameError"></span>

<label>Password: <input type="password" id="password" required></label>

<span class="error" id="passwordError"></span>

<label>Email: <input type="email" id="email" required></label>

<span class="error" id="emailError"></span>

<label>Mobile No: <input type="text" id="mobile" required></label>

<span class="error" id="mobileError"></span>

<label>City:

<select id="city" required>

<option value="Select">Select</option>

<option value="New York">New York</option>

<option value="Los Angeles">Los Angeles</option>

<option value="Chicago">Chicago</option>

</select>

</label>

<span class="error" id="cityError"></span>

<button type="submit">Register</button>

</form>

<a href="index.html">Back to Home</a>

</div>

<div class="container symbol-table">

<h2>Symbol Table</h2>

<table id="symbolTable">

<thead>

<tr>

<th>Field Name</th>

<th>Type</th>

<th>Validation Regex</th>

</tr>

</thead>

<tbody>

<!-- Symbol table entries will be added here -->

</tbody>

</table>

</div>

<script>

const form = document.getElementById('registrationForm');

const symbolTable = [

{ fieldName: 'firstName', type: 'text', regex: /^[A-Za-z]+$/ },

{ fieldName: 'lastName', type: 'text', regex: /^[A-Za-z]+$/ },

{ fieldName: 'username', type: 'text', regex: /^[A-Za-z0-9\_]{3,16}$/ },

{ fieldName: 'password', type: 'password', regex: /^(?=.\*[A-Z])(?=.\*[a-z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/ },

{ fieldName: 'email', type: 'email', regex: /^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/ },

{ fieldName: 'mobile', type: 'text', regex: /^\d{10}$/ },

{ fieldName: 'city', type: 'select', regex: /^(Select|New York|Los Angeles|Chicago)$/ }

];

const populateSymbolTable = () => {

const tableBody = document.getElementById('symbolTable').querySelector('tbody');

symbolTable.forEach(entry => {

const row = document.createElement('tr');

Object.values(entry).forEach(value => {

const cell = document.createElement('td');

cell.textContent = value.toString();

row.appendChild(cell);

});

tableBody.appendChild(row);

});

};

form.addEventListener('submit', function(event) {

// Regular Expressions

const nameRegex = /^[A-Za-z]+$/;

const usernameRegex = /^[A-Za-z0-9\_]{3,16}$/;

const passwordRegex = /^(?=.\*[A-Z])(?=.\*[a-z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/;

const emailRegex = /^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

const mobileRegex = /^\d{10}$/;

const cityRegex = /^(Select|New York|Los Angeles|Chicago)$/;

// Input Values

const firstName = document.getElementById('firstName').value;

const lastName = document.getElementById('lastName').value;

const username = document.getElementById('username').value;

const password = document.getElementById('password').value;

const email = document.getElementById('email').value;

const mobile = document.getElementById('mobile').value;

const city = document.getElementById('city').value;

// Error Messages

const firstNameError = document.getElementById('firstNameError');

const lastNameError = document.getElementById('lastNameError');

const usernameError = document.getElementById('usernameError');

const passwordError = document.getElementById('passwordError');

const emailError = document.getElementById('emailError');

const mobileError = document.getElementById('mobileError');

const cityError = document.getElementById('cityError');

// Reset Error Messages

firstNameError.textContent = '';

lastNameError.textContent = '';

usernameError.textContent = '';

passwordError.textContent = '';

emailError.textContent = '';

mobileError.textContent = '';

cityError.textContent = '';

// Validation

let isValid = true;

if (!nameRegex.test(firstName)) {

firstNameError.textContent = 'Invalid First Name';

isValid = false;

}

if (!nameRegex.test(lastName)) {

lastNameError.textContent = 'Invalid Last Name';

isValid = false;

}

if (!usernameRegex.test(username)) {

usernameError.textContent = 'Invalid Username';

isValid = false;

}

if (!passwordRegex.test(password)) {

passwordError.textContent = 'Invalid Password';

isValid = false;

}

if (!emailRegex.test(email)) {

emailError.textContent = 'Invalid Email';

isValid = false;

}

if (!mobileRegex.test(mobile)) {

mobileError.textContent = 'Invalid Mobile Number';

isValid = false;

}

if (!cityRegex.test(city)) {

cityError.textContent = 'Invalid City';

isValid = false;

}

if (!isValid) {

event.preventDefault();

} else {

alert('Registration Successful!');

}

});

// Populate the symbol table on page load

populateSymbolTable();

</script>

</body>

</html>

**Output:**

