|  |  |
| --- | --- |
|  | **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: | | **Talha Azeem** | | Registration #: | | | **SP20-bcs-047** | | |
|  | |  | |  | | |  | | |

**Question 1**

Write an introduction of your compiler construction project.

**Answer:**

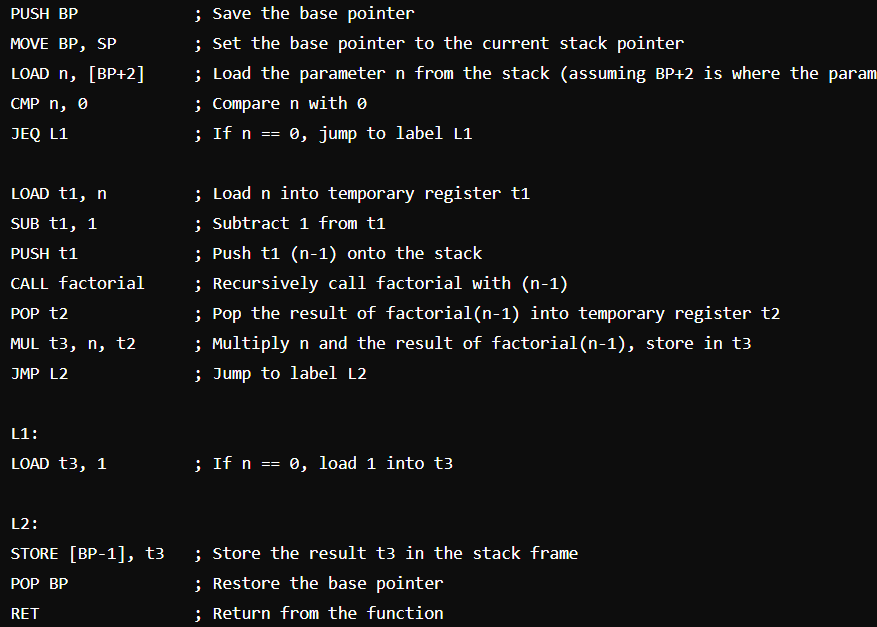
The key components of the project are as follows:

1. **Lexical Analysis (Scanning)**: The first phase involves breaking down the source code into tokens using a lexer or scanner. Tokens are the basic building blocks of the language, such as keywords, operators, identifiers, and literals.
2. **Syntax Analysis (Parsing)**: The parser takes the tokens generated by the lexer and organizes them into a syntax tree (also known as a parse tree). This tree represents the grammatical structure of the source code according to the rules of the MiniLang grammar.
3. **Semantic Analysis**: This phase ensures that the syntax tree follows the semantic rules of the language. It involves type checking, scope resolution, and ensuring that operations are semantically correct.
4. **Intermediate Code Generation**: The validated syntax tree is translated into an intermediate representation, which is a lower-level code that is easier to optimize and translate into machine code.
5. **Optimization**: The intermediate code is optimized for performance improvements. This includes eliminating redundant code, optimizing loops, and other code enhancement techniques.
6. **Code Generation**: The final phase involves translating the optimized intermediate code into target machine code or bytecode that can be executed by a virtual machine or hardware processor.
7. **Error Handling**: Throughout the compilation process, the compiler will detect and report errors in the source code, providing meaningful feedback to help developers correct their programs.

**Question 2**

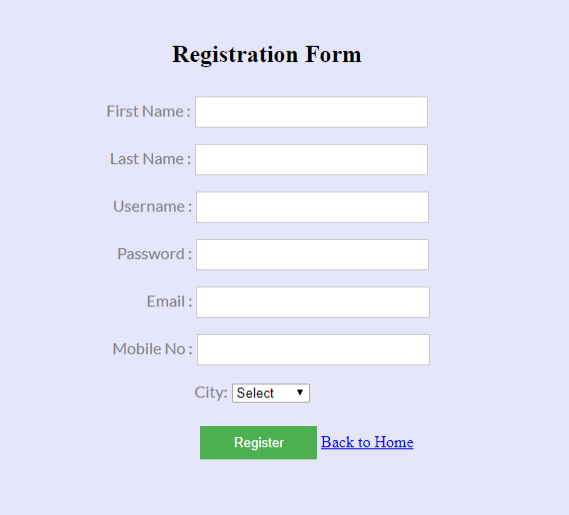
Give a sample input and output for your compiler construction project?

**Answer:**



**Question 3**

Create and implement RE and DFAs for the form below



You must use Regex to validate data.

**Answer:**

Code:  
import re

# Define Regular Expressions

patterns = {

    "first\_name": re.compile(r"^[A-Za-z]+$"),

    "last\_name": re.compile(r"^[A-Za-z]+$"),

    "username": re.compile(r"^[A-Za-z0-9\_]+$"),

    "password": re.compile(r"^(?=.\*[A-Za-z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$"),

    "email": re.compile(r"^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$"),

    "mobile\_no": re.compile(r"^\d{10}$"),

}

# Sample Inputs

inputs = {

    "first\_name": "John",

    "last\_name": "Doe",

    "username": "john\_doe",

    "password": "P@ssw0rd",

    "email": "john.doe@example.com",

    "mobile\_no": "1234567890",

}

# Validate Inputs

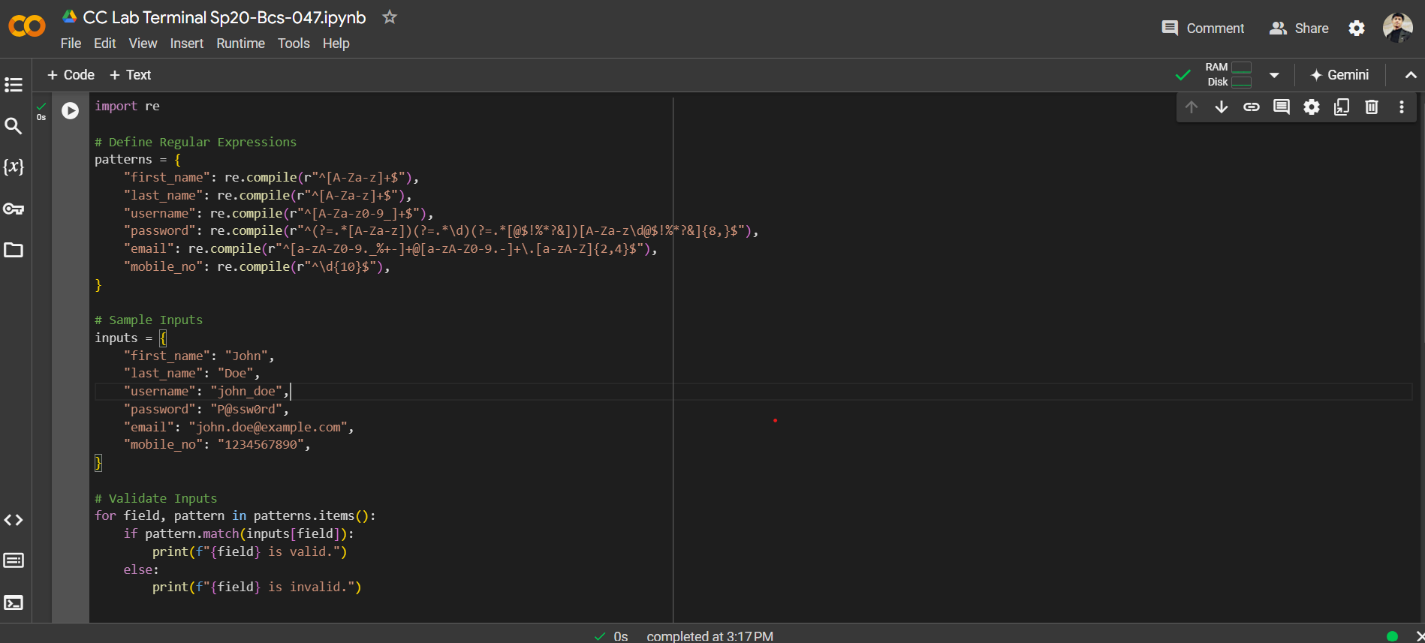
for field, pattern in patterns.items():

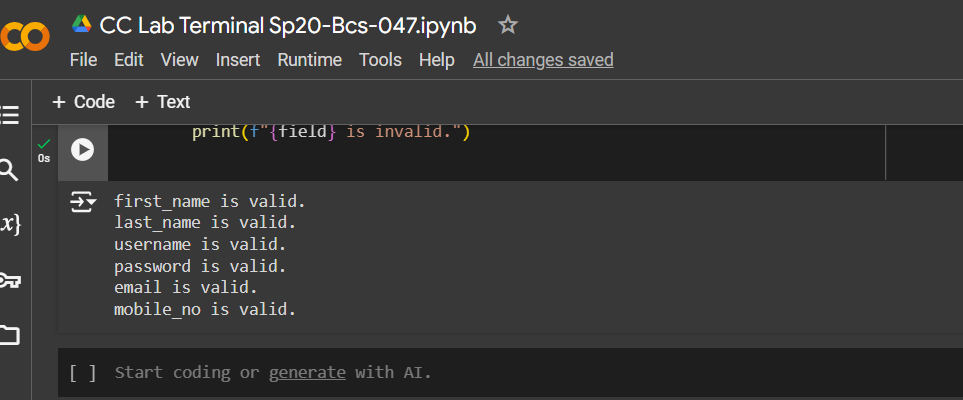
    if pattern.match(inputs[field]):

        print(f"{field} is valid.")

    else:

        print(f"{field} is invalid.")





**Question 4:**

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

**Answer:**

import re

class SymbolTableEntry:

    def \_\_init\_\_(self, identifier, type\_, scope, attributes=None):

        self.identifier = identifier

        self.type\_ = type\_

        self.scope = scope

        self.attributes = attributes if attributes else {}

    def \_\_repr\_\_(self):

        return f"{self.identifier}: {{'type': '{self.type\_}', 'scope': '{self.scope}', 'attributes': {self.attributes}}}"

class SymbolTable:

    def \_\_init\_\_(self):

        self.table = []

    def add\_entry(self, entry):

        self.table.append(entry)

    def \_\_repr\_\_(self):

        return "\n".join(str(entry) for entry in self.table)

# Define MiniLang code

mini\_lang\_code = """

function factorial(n) {

    if (n == 0) {

        return 1;

    } else {

        return n \* factorial(n - 1);

    }

}

let result = factorial(5);

print(result);

"""

# Regular expressions to capture function definitions, variable declarations, and function calls

function\_def\_re = re.compile(r"function\s+(\w+)\s\*\((.\*?)\)\s\*{")

variable\_decl\_re = re.compile(r"let\s+(\w+)\s\*=\s\*(.+?);")

function\_call\_re = re.compile(r"(\w+)\s\*\((.\*?)\);")

# Symbol table

symbol\_table = SymbolTable()

# Parsing the MiniLang code

lines = mini\_lang\_code.split('\n')

scope = "global"

for line in lines:

    # Check for function definitions

    function\_def\_match = function\_def\_re.search(line)

    if function\_def\_match:

        function\_name = function\_def\_match.group(1)

        parameters = function\_def\_match.group(2).split(',')

        parameters = [param.strip() for param in parameters if param.strip()]

        symbol\_table.add\_entry(SymbolTableEntry(function\_name, "function", scope, {"parameters": parameters}))

        scope = function\_name

        continue

    # Check for variable declarations

    variable\_decl\_match = variable\_decl\_re.search(line)

    if variable\_decl\_match:

        var\_name = variable\_decl\_match.group(1)

        var\_value = variable\_decl\_match.group(2)

        symbol\_table.add\_entry(SymbolTableEntry(var\_name, "variable", scope, {"value": var\_value}))

        continue

    # Check for function calls

    function\_call\_match = function\_call\_re.search(line)

    if function\_call\_match:

        func\_name = function\_call\_match.group(1)

        arguments = function\_call\_match.group(2).split(',')

        arguments = [arg.strip() for arg in arguments if arg.strip()]

        symbol\_table.add\_entry(SymbolTableEntry(func\_name, "function\_call", scope, {"arguments": arguments}))

        continue

print("Symbol Table:")

print(symbol\_table)

