



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: Haseeb Hassan

Registration #: SP20-bcs-031

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:

```
PUSH BP          ; Save the base pointer
MOVE BP, SP      ; Set the base pointer to the current stack pointer
LOAD n, [BP+2]   ; Load the parameter n from the stack (assuming BP+2 is where the param
CMP n, 0         ; Compare n with 0
JEQ L1          ; If n == 0, jump to label L1


LOAD t1, n       ; Load n into temporary register t1
SUB t1, 1        ; Subtract 1 from t1
PUSH t1          ; Push t1 (n-1) onto the stack
CALL factorial   ; Recursively call factorial with (n-1)
POP t2           ; Pop the result of factorial(n-1) into temporary register t2
MUL t3, n, t2    ; Multiply n and the result of factorial(n-1), store in t3
JMP L2          ; Jump to label L2

L1:
LOAD t3, 1       ; If n == 0, load 1 into t3

L2:
STORE [BP-1], t3 ; Store the result t3 in the stack frame
POP BP          ; Restore the base pointer
RET            ; Return from the function
```

Question 3

Create and implement RE and DFAs for the form below



The image shows a registration form titled "Registration Form" in bold black text. Below the title, there are six input fields: "First Name :", "Last Name :", "Username :", "Password :", "Email :", and "Mobile No :". Each field is a white rectangular box with a thin border. Below the "Mobile No :" field, there is a "City:" label followed by a dropdown menu with "Select" and a downward arrow. At the bottom of the form, there is a green rectangular button labeled "Register" and a blue text link labeled "Back to Home".

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}\$"),

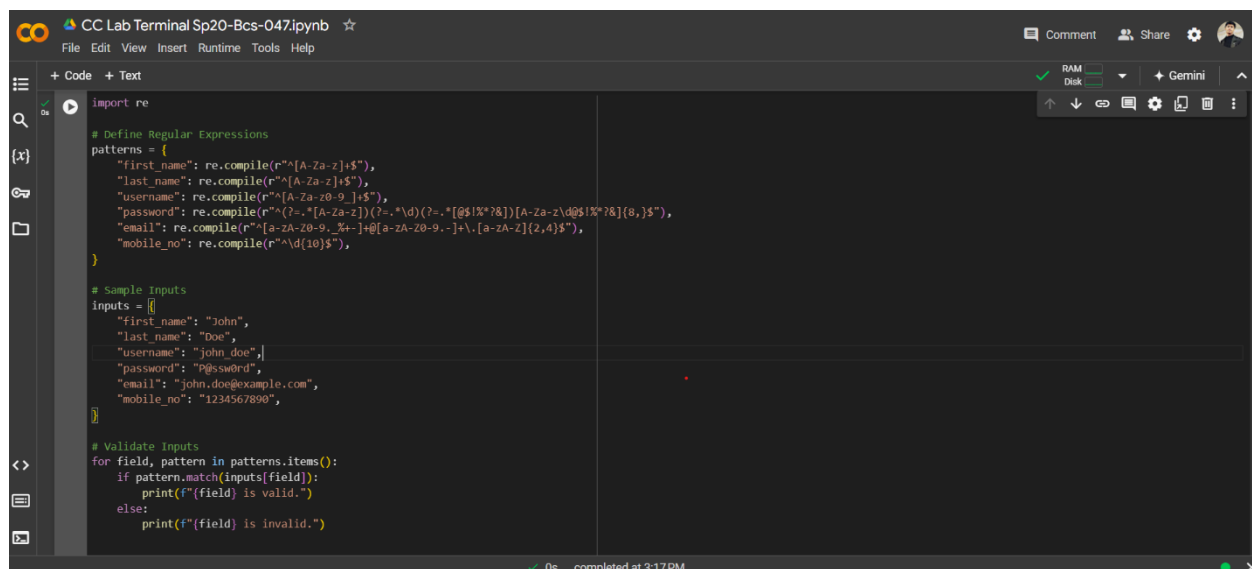
```
}
```

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

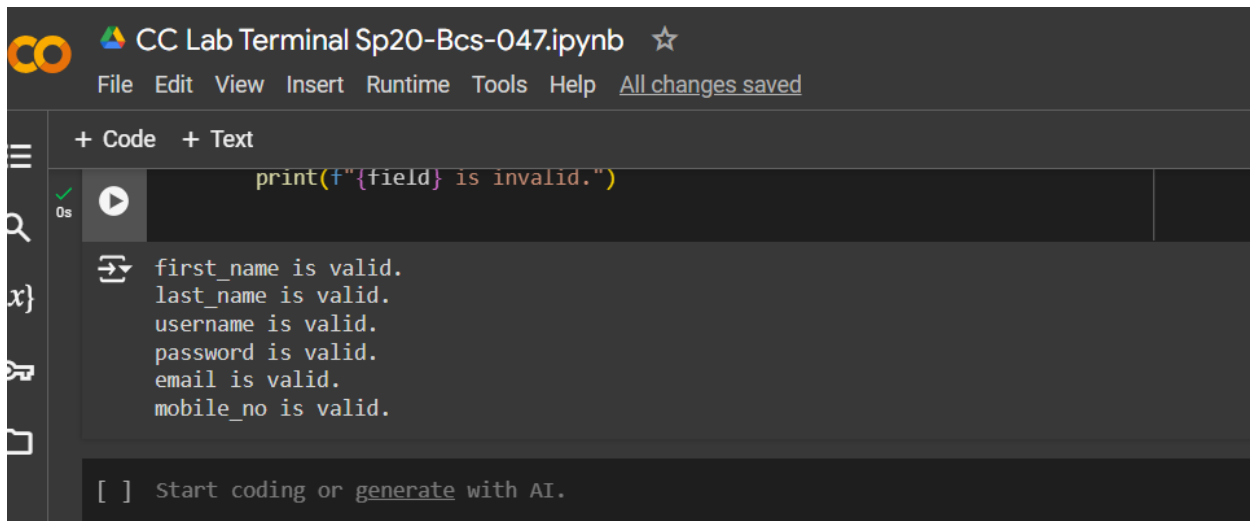
```
for field, pattern in patterns.items():  
    if pattern.match(inputs[field]):  
        print(f"{field} is valid.")  
    else:  
        print(f"{field} is invalid.")
```



The screenshot shows a Jupyter Notebook titled "CC Lab Terminal Sp20-Bcs-047.ipynb". The code is written in Python and defines regular expressions for validating various input fields. The code is as follows:

```
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.")
```

The notebook interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for running, saving, and other actions, and a status bar at the bottom indicating the code was "completed at 3:17 PM".



The screenshot shows a Jupyter Notebook window titled "CC Lab Terminal Sp20-Bcs-047.ipynb". The menu bar includes File, Edit, View, Insert, Runtime, Tools, Help, and a link for "All changes saved". The interface has a sidebar on the left with icons for a menu, search, and file explorer. The main area shows a code cell with the text `print(f"{field} is invalid.")`. Below the code cell, the output is displayed: `first_name is valid.`, `last_name is valid.`, `username is valid.`, `password is valid.`, `email is valid.`, and `mobile_no is valid.`. At the bottom of the notebook, there is a prompt: `[] Start coding or generate with AI.`

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(',')
```

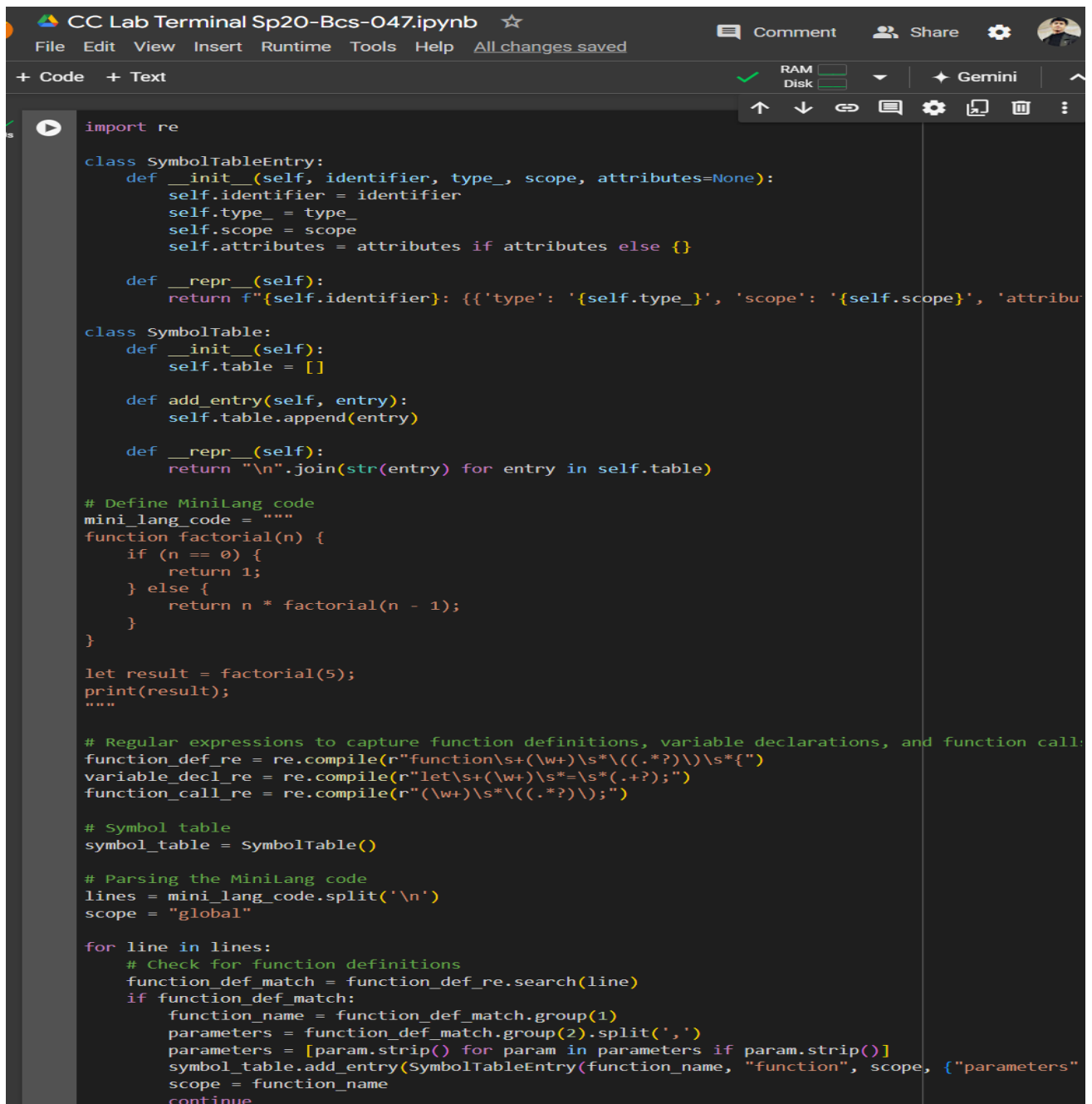
```
        arguments = [arg.strip() for arg in arguments if arg.strip()]
```

```
symbol_table.add_entry(SymbolTableEntry(func_name, "function_call", scope, {"arguments":  
arguments})))
```

```
continue
```

```
print("Symbol Table:")
```

```
print(symbol_table)
```



The screenshot shows a Jupyter Notebook titled "CC Lab Terminal Sp20-Bcs-047.ipynb". The code defines a `SymbolTableEntry` class and a `SymbolTable` class. The `SymbolTable` class has methods `__init__`, `add_entry`, and `__repr__`. Below the class definitions, there is a MiniLang code snippet for a factorial function, its execution, and a parsing routine that uses regular expressions to extract function definitions from the MiniLang code and adds them to the symbol table.

```
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}', 'attribu"

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 call:
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"
        scope = function_name
        continue
```



```

        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)

```

```

Symbol Table:
factorial: {'type': 'function', 'scope': 'global', 'attributes': {'parameters': ['n']}}
factorial: {'type': 'function_call', 'scope': 'factorial', 'attributes': {'arguments': ['n - 1']}}
result: {'type': 'variable', 'scope': 'factorial', 'attributes': {'value': 'factorial(5)'}}
print: {'type': 'function_call', 'scope': 'factorial', 'attributes': {'arguments': ['result']}}

```


CC Lab Terminal Sp20-Bcs-047.ipynb
☆


File Edit View Insert Runtime Tools Help [All changes saved](#)

+ Code + Text

```

print(symbol_table)
print(symbol_table)

```


0s

```

Symbol Table:
factorial: {'type': 'function', 'scope': 'global', 'attributes': {'parameters': ['n']}}
factorial: {'type': 'function_call', 'scope': 'factorial', 'attributes': {'arguments': ['n - 1']}}
result: {'type': 'variable', 'scope': 'factorial', 'attributes': {'value': 'factorial(5)'}}
print: {'type': 'function_call', 'scope': 'factorial', 'attributes': {'arguments': ['result']}}

```

☐ Start coding or [generate](#) with AI.