Final Project

Part A:

In this project, you will develop a new functional programming interpreter, according the requirements below.

Objective:
Develop an interpreter for a simple functional programming language that emphasizes function definitions and lambda expressions, without variable assignments or mutable state.

Language Specifications:

1. Data Types:
   - INTEGER: Whole numbers (e.g., -3, 0, 42)
   - BOOLEAN: True and False

2. Operations:
   a. Arithmetic Operations (for INTEGERs):
      - Addition (+)
      - Subtraction (-)
      - Multiplication (*)
      - Division (/) (integer division)
      - Modulo (%)

   b. Boolean Operations:
      - AND (&&)
      - OR (||)
      - NOT (!)

   c. Comparison Operations:
      - Equal to (==)
      - Not equal to (!=)
      - Greater than (>)
      - Less than (<)
      - Greater than or equal to (>=)
      - Less than or equal to (<=)

3. Functions:
   - Named function definitions
   - Anonymous functions (lambda expressions)
   - Function application

4. Recursion:

   - Support for recursive function calls
- It should support a replacement for while loop.

5. Immutability:
   - All values are immutable
   - No variable assignments or state changes

Project Requirements:

1. Lexer:
   - Implement a lexer that tokenizes the input source code
   - Handle whitespace, comments, and all language constructs

2. Parser:
   - Implement a parser that creates an Abstract Syntax Tree (AST) from the tokenized input
- The parser should use the BNF of the language. It should read it and parsing the commands according the BNF definition.
   - Handle function definitions, lambda expressions, and all language constructs

3. Interpreter:
   - Each program in this language should have a suffix *.lambda
 - The interpreter will be able to execute commands in both interactive mode (line by line) or executing a full program. In both cases, it will print the result of each command after its execution.
   - Implement an interpreter that evaluates the AST
   - Support function application, including higher-order functions
   - Implement a call stack for managing function calls and recursion

4. Error Handling:
   - Implement comprehensive error checking and reporting
   - Provide meaningful error messages for syntax errors, type errors, and runtime errors

5. REPL (Read-Eval-Print Loop):
   - Implement a REPL for interactive use of the language

6. Documentation:
   - Provide the BNF (Backus–Naur form) grammar
   - Provide clear documentation for the language syntax and features
   - Include comments in the code explaining key components and algorithms

- Write a document describing all of the design considerations and assumptions relating to that section of the language.

8. Testing:
   - Develop a comprehensive test suite covering all language features
   - Include edge cases and error conditions in the tests

Sample Language Syntax:

```
# Function definition
Defun {'name':  'factorial', 'arguments': (n,)}
   (n == 0) or  (n * factorial(n - 1))

# Lambda expression
(Lambd x.(Lambd  y. (x + y)))

# Function application
factorial(5)

# Boolean operations
(x > 0) && (y < 10)

# Arithmetic operations
(3 + 4) * (2 - 1)
```

Evaluation Criteria:

1. Correctness: The interpreter correctly implements all language features.
2. Code Quality: Well-structured, readable, and well-commented code.
3. Error Handling: Robust error detection and reporting.
4. Efficiency: Reasonable performance for interpreting programs.
5. Documentation: Clear and comprehensive documentation of the language and interpreter.
6. Testing: Thorough test coverage demonstrating the correct implementation of all features.

Submission Requirements:

1. Well-documented Python source code repository in GitHub. Use proper code structuring, naming conventions.

2. Documentation of the BNF, detailed language syntax and features. The document should describe in details the BNF grammar, trade-offs, and limitations of the language.
3. User guide for running the interpreter in both modes
4. Test suite demonstrating all implemented features. Write a program with 30 lines that demonstrates the capabilities of the language. Examples of code written in your implemented languages. Include test cases. This program should demonstrate how to execute "while loop" with recursion calls.
5. A report discussing the design decisions, challenges faced, and solutions implemented
6. Answers to all theoretical questions.
7. After the project submission, each group will have to schedule 15 minutes meeting with me to demonstrate the project and answer some questions.

Part B:

1.) Implement a Fibonacci sequence generator using a single lambda expression that returns a list of the first n Fibonacci numbers. The function should take n as an input.

2.) Write the shortest Python program, that accepts a list of strings and return a single string that is a concatenation of all strings with a space between them. Do not use the "join" function. Use lambda expressions.

3.) Write a Python function that takes a list of lists of numbers and return a new list containing the cumulative sum of squares of even numbers in each sublist. Use at least 5 nested lambda expressions in your solution.

4.) Write a higher-order function that takes a binary operation (as a lambda function) and returns a new function that applies this operation cumulatively to a sequence. Use this to implement both factorial and exponentiation functions.

5) Rewrite the following program in one line by using nested filter, map and reduce functions:

```
nums = [1,2,3,4,5,6]

evens = []

for num in nums:

   if num % 2 == 0:

      evens.append(num)
```

```
squared = []
for even in evens:
    squared.append(even**2)

sum_squared = 0
for x in squared:
    sum_squared += x

print(sum_squared)
```

6.) Write one-line function that accepts as an input a list of lists containing strings and returns a new list containing the number of palindrome strings in each sublist. Use nested filter / map / reduce functions.

7.) Explain the term "lazy evaluation" in the context of the following program:

```
def generate_values():
  print('Generating values...')
  yield 1
  yield 2
  yield 3

def square(x):
  print(f'Squaring {x}')
  return x * x

print('Eager evaluation:')
values = list(generate_values())
squared_values = [square(x) for x in values]
print(squared_values)
print('\nLazy evaluation:')

squared_values = [square(x) for x in generate_values()]

print(squared_values)
```

8.) Write a one-line Python function that takes a list of integers and returns a new list containing only the prime numbers, sorted in descending order. Use lambda expressions and list comprehensions.