Final Project

Part A:

In this project, you will define and implement programming languages with different characteristics. For each section, you must:

- Provide the BNF (Backus–Naur form) grammar, if it does not change from the previous section explicitly writing it out.
- Implement the language by writing code in Python, Java, and C.
- Write a document describing all of the design considerations and assumptions relating to that section of the language.

The project deliverables should include:

- Well-documented code repositories in GitHub for the Python, Java, and C implementations. Use proper code structuring, naming conventions, and comments.
- A report detailing the BNF grammar, design decisions, trade-offs, and limitations in each implementation. Discuss how the languages evolve through each milestone.
- Examples of code written in your implemented languages. Include test cases.
- Instructions on how to run and compile your code, as well as explanations of the outputs.
- Answers to all theoretical questions.
- After the submission of the project, each group will have to schedule 15 minutes meeting with me to demonstrate the project and answer some questions.

The goal of this project is to gain both hands-on experience and theoretical understanding of programming languages and what is behind them.

1. Design an interpreter for a new scripting language that can perform simple integer arithmetic operations including addition, subtraction, multiplication, and integer division. Design the syntax, variables, data types and control flows.
2. Provide the complete BNF grammar for the language. Update the grammar as features are added in each milestone:
3. Implement memory management:
   - Define the maximum length allowed for calculation results and program code.
   - Specify overflow handling if these lengths are exceeded.

- Declare variable naming rules and maximum number of variables allowed.

4. Is your language is statically typed or dynamically typed? Explain in details your choice.
5. Add support for Boolean expressions using comparison operators >, <, ==.
6. Implement if-then conditional statements without else clauses. Allow nested conditionals up to 3 levels deep.
7. Add while loops with conditional continuation. Support loop bodies containing nested conditionals up to 3 levels deep.
8. Demonstrate the capabilities of the language by implementing an intricate program up to 50 lines long. This program should utilize arithmetic, variables, comparisons, conditional logic, and looping to showcase the complexity possible with your language design.

For each of the above sections provide:

- The updated BNF grammar
- Code implementing the interpreter in Python
- Example programs and test cases
- Discussion of design decisions and trade-offs

Your language should be from the Functional Paradigm.

Part B:

9.) Implement a factorial function in one line by using lambda expressions.

10.) 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.

11.) 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.

12.) 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)
```

13.) 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.

14.) 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)
```