**Final Project - 316594829 Oren Golovchik**

**Part A:**

• Is your language is statically typed or dynamically typed? Explain in details

your choice.

The code represents a dynamically typed language.

The code doesn't ask for the type of variables upfront and just figures out what they are as it goes along. Like, you don't have to say "this is a number" or "this is text," it just knows. Also, when it runs through the code, it doesn't check if what you're doing makes sense in terms of types before doing it. It just tries to do it and sees what happens. For instance, in the execute\_statement part of the Interpreter class, it looks at what's there and deals with it on the spot without worrying about what type it is beforehand.

• Demonstrate the capabilities of the language by implementing an intricate

program up to 50 lines long - Discussion of design decisions and trade-offs.

Design Decisions:

The Fibonacci code iteratively calculates Fibonacci numbers up to a certain limit and then identifies prime Fibonacci numbers among them.

It uses a straightforward approach of calculating Fibonacci numbers iteratively using a while loop.

It checks for prime numbers within the Fibonacci sequence using a basic primality check.

Trade-offs:

This implementation sacrifices memory efficiency for simplicity by storing all Fibonacci numbers in a list. For very large inputs, this could lead to high memory consumption.

•For each milestone in the interpreter language, describe how converting to a compiled language affects the design and implementation.

Milestone 1: Basic Arithmetic Operations:

In a compiled language, the design would focus on generating machine code or intermediate representation directly from the provided grammar rules. This would involve building an abstract syntax tree and translating it into executable code.

Milestone 2: Variables, Data Types, and Control Flows:

Compilation would involve generating code to handle variables, data types, and control flows efficiently. This includes managing memory allocation, optimizing variable access, and generating control flow instructions.

Milestone 3: Memory Management and Boolean Expressions:

Compilation would involve implementing memory management strategies such as garbage collection or manual memory allocation/deallocation. Boolean expressions would need to be translated into machine-level instructions for evaluation.

**Part B:**

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)

With lazy evaluation, we don't immediately do all the work. Instead, we wait until we need each number before generating it and squaring it. So, we only doing the work when it's necessary, not beforehand.

The code demonstrates two types of evaluation strategies: eager evaluation and lazy evaluation:

In the code, eager evaluation is demonstrated by first creating a list of values using list(generate\_values()), and then applying the square() function to each value in the list comprehensively to produce a list of squared values. This means that all values from generate\_values() are generated upfront, and then their squares are calculated sequentially.

Lazy evaluation, on the other hand, defers the computation until it is actually needed. In the code the generate\_values() function is a generator that yields three values, these values are not generated immediately upon calling generate\_values(). Instead, they are produced one at a time as the list comprehension iterates over them. The square(x) function is applied to each value as it is generated.  
This means that the values are generated one at a time, and their squares are calculated immediately after each value is generated.