Section 1: Program

Please include a zip file of the final repository in this directory.

Section 2: Teamwork

I personally did all of the coding that went into the source.zip folder, my teamate covered documentation and gave me 3 tests to test my code. Test cases

1.
```
assertEquals(60, executeProgram(function test() : int {
            var y = 120
            var x = 2
            var n = y / x
            return n}
            print(test())
```

2.
```
assertEquals("Hellonull", executeProgram(function test(x,y) : String {
            var output = x  + y
            return output }
            print(test("Hello",null)
```

3.
```
 assertEquals(ar\nbr\ncr\n, executeProgram(for x in ["a","b","c"] { var z = "r"
                                                    print(x + z)}
```

Section 3: Design pattern

Memoization was used in creating the List Types when creating a list literal. The function creates a single list type that is stored rather than creating several instances of types for each element. It also means that every element will evaluate to that type thus also stopping the need to re-initialize every element's type. Saving the need for many instances.

Section 4: Technical writing. Include the technical

document that accompanied your capstone project.

Section 5: UML.

for (x in [1,2,3]) { print(x)}

This is a for loop sequence diagram citing how the For Loop Statement is parsed, parseProgram takes an input, calls parseStatement

which then will return null until calling parseForLoop, which will then again call parseStatement, again returning null until it calls parsePrintStatement which will then call parseExpression UML Diagram is in the folder as a PNG would not work here for whatever reason.

Section 6: Design trade-offs

While Recursive Descent Parsing you have much more control over the parsing, it is also much more readable. While The syntax is not only straight forward, it's much easier to understand what is happening at evaluation and parsing making it a better learning tool. While it is a better tool for learning it is not the better tool for performance. It struggles for difficult grammar or complex parsing, this along with the fact that Byte Code is available across multiple platforms without having to change the code. With that said JVM SUCKED. The choice between using a Byte Code Generator or Recursive Descent Parsing depends heavily on the specific requirements of the programming task. If the goal is to build a portable

and efficient executable program, a byte code generator is obviously better. This being said, if the task requires a straightforward implementation with a focus on ease of maintenance and clarity, recursive descent parsing is the better choice.

Section 7: Software development life cycle model

Describe the model that you used to develop your capstone project. How did this model help and/or hinder your team?

We are using Test Driven Development (TDD) for this project Test Driven Development (TDD) is a development approach where tests are written before the source code. The idea is to create tests that will show success when the desired output is repeated to the expected. This cycle usually utalizes a fair of tests, and is not done one test, write, another test. Though the amount of tests are applicable to small snippits of code to create a path of progression.TDD makes sure that every section of code is checked. We used TDD for every graded checkpoint of creating CatScript. This lead to a feeling of progression that made it much easier for me to progress as every checkpoint had several subsections of test leading to constant dopamine and making it manageable. Critics of this model would say that it creates a culture of over testing for simple tasks, I would argue that has nothing to do with the approach and more to do with poor management. I completely see how this could be abused in a corperate environment, but works well for education.