Section 1:

https://github.com/RichardBabcock/csci-468-spring2025-private/blob/main/capstone/portfolio/source.zip (https://github.com/RichardBabcock/csci-468-spring2025-private/blob/main/capstone/portfolio/source.zip)

Section 2:

```
Me and my partner started off with bouncing ideas back and forth about how to start this team work section. We decided to start with
We then proceed to create three new tests each my partner decided to take our known tests and combine bits and pieces of them togeth
void forLoopInsideFunctionWorksProperly() {
    assertEquals("1\n2\n3\n", executeProgram("function foo() { for(x in [1, 2, 3]) { print(x) } }\n" +
            "foo()\n"
    ));
}

public void elseStatementEnsuresOpeningBrace() {
    IfStatement expr = parseStatement("if(x > 10){ print(x) } else } ", false);
    assertNotNull(expr);
    assertTrue(expr.hasErrors());
}

void nestedIfStatementWorksProperly() {
    assertEquals("2\n3\n", executeProgram("if(true){ if(true){ print(2) } else { print(1) } if(true){ print(3) } else { print(1) } }"
}


As you can see they are more complex and expanded the testing of the parser compared to what the given tests were.
```

Section 3:

```
The design pattern that was used in my capstone was memoization. It was used in the CatscriptType file starting at line 34 and used
```

Section 4:

Catscript Guide

This document is a guide for catscript, to satisfy capstone requirement 4

1. Introduction

Catscript is a simple statically-typed scripting language. It is lightweight and supports basic data structures and control flow Here is an example of Catscript code:

```
var x = "foo"
print(x)
```

2. Types

Catscipt contains a small type system that is as follows:

```
* int - a 32 bit integer
* string - a java-style string
* bool - a boolean value
* list<x> - a list of values with the type 'x'
* null - the null type
* object - any type of value
```

3. Variables and Assignments

3.1 Variable Statements

```
var x = 1
```

3.2 Assignment Statements

```
x = 2
```

Types are mandatory upon declaration, but are in local scope if the type is not ambiguous.

4. Control Flow

4.1 For loops

```
for( x in [1, 2, 3] ) {
    var y = x
    print(y)
}
```

4.2 If Statements

```
if(true){
    print("true")
} else{
    print("false")
}
```

5. Functions

5.1 Function Call Statements

```
function foo() : int{
    var x = 10
    return x
    }

 print( foo() )
```

5.2 Return Statements

```
function foo(x : int){
    return x
    }
```

5.3 Print Statements

```
print("Hello World!")
```

6. Operators

6.1 Arithmetic:

```
+, -, *, /
```

6.2 Comparison:

```
==, !=, <, <=, >, >=
```

6.3 Unary:

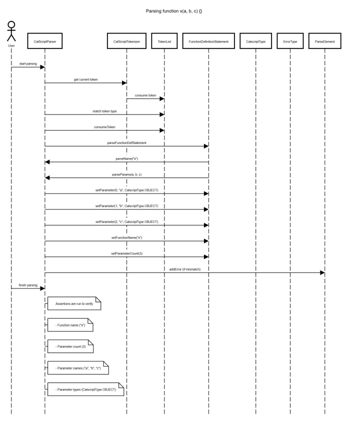
```
not, -
```

String concatenation using + is also allowed.

7. Grammar

```
catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            return_statement |
            function_call_statement;

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
                '{', { statement }, '}';

if_statement = 'if', '(', expression, ')', '{',
                    { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];

print_statement = 'print', '(', expression, ')'

variable_statement = 'var', IDENTIFIER,
     [':', type_expression, ] '=', expression;

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                       [ ':' + type_expression ], '{',  { statement },  '}';

parameter_list = [ parameter, {',' parameter } ];

parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [, expression];

expression = equality_expression;

equality_expression = comparison_expression { ("!=" | "==") comparison_expression };

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };

additive_expression = factor_expression { ("+" | "-" ) factor_expression };

factor_expression = unary_expression { ("/" | "*" ) unary_expression };

unary_expression = ( "not" | "-" ) unary_expression | primary_expression;

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null"|
                     list_literal | function_call | "(", expression, ")"

list_literal = '[', expression,  { ',', expression } ']';

function_call = IDENTIFIER, '(', argument_list , ')'

argument_list = [ expression , { ',' , expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression, '>']
```

Section 5:


Parsing function x(a, b, c) {}
User
CatScriptParser
CatScriptTokenizer
TokenList
FunctionDefinitionStatement
CatscriptType
ErrorType
ParseElement
start parsing
get current token
consume token
match token type
consumeToken
parseFunctionDefStatement
parseName("x")
parseParams(a, b, c)
setParameter(0, "a", CatscriptType.OBJECT)
setParameter(1, "b", CatscriptType.OBJECT)
setParameter(2, "c", CatscriptType.OBJECT)
setFunctionName("x")
setParameterCount(3)
addError (if mismatch)
finish parsing
Assertions are run to verify
- Function name ("x")
- Parameter count (3)
- Parameter names ("a", "b", "c")
- Parameter types (CatscriptType.OBJECT)

Section 6:

In this class we wrote our parser using recursive descent, this gave us some insights that we would not have had compared to using a
However, while recursive descent has it’s merits, it also has drawbacks. It doesn’t provide the same level of control over performan

Section 7:

The life cycle model employed in the Compilers class is Test-Driven Development (TDD). My experience with TDD has significantly bene