Section 1: Program

The developed program can be found in .\csci-468-spring2024-private\capstone\portfolio\source.zip

Section 2: Teamwork

For this project I had one partner, His main contribution was providing me the Catscript documentation, as well as creating three tests to run on my code so that I could find some bugs in the code base. His tests were very helpful because they helped me realize that I had been parsing out a few different types of statements and expressions incorrectly. The following code is the tests that my partner produced for me.

```
void testingListTypes(){
        VariableStatement var = parseStatement("var x : list<int> = [1]");
        assertEquals(CatscriptType.getListType(CatscriptType.INT),
var.getExplicitType());

        VariableStatement var1 = parseStatement("var x : list<bool> = [true]");
        assertEquals(CatscriptType.getListType(CatscriptType.BOOLEAN),
var1.getExplicitType());

        VariableStatement var2 = parseStatement("var x : list<string> =
[\"foo\"]");
        assertEquals(CatscriptType.getListType(CatscriptType.STRING),
var2.getExplicitType());

        VariableStatement var3 = parseStatement("var x : list<object> = [1]");
        assertEquals(CatscriptType.getListType(CatscriptType.OBJECT),
var3.getExplicitType());
    }

    @Test
    void testingElseIfStatements() {
        assertEquals("21\n", executeProgram("function foo(x : int) : int {\n" +
                "   if (x < 21) {\n" +
                "       return x * 2 }\n" +
                "   else if (x == 21) {\n" +
                "       return x }\n" +
                "   else { return -x }" +
                "   return 0 }\n" +
                "var a : int = 21\n" +
                "print(foo(a))"
        ));
        assertEquals("40\n", executeProgram("function foo(x : int) : int {\n" +
                "   if (x < 21) {\n" +
                "       return x * 2 }\n" +
                "   else if (x == 21) {\n" +
                "       return x }\n" +
                "   else { return -x }" +
```

```
                "   return 0 }\n" +
                "var a : int = 20\n" +
                "print(foo(a))"
        ));
        assertEquals("-23\n", executeProgram("function foo(x : int) : int {\n" +
                "   if (x < 21) {\n" +
                "       return x * 2 }\n" +
                "   else if (x == 21) {\n" +
                "       return x }\n" +
                "   else { return -x }" +
                "   return 0 }\n" +
                "var a : int = 23\n" +
                "print(foo(a))"
        ));
    }

    @Test
    void testingVars() {
        assertEquals("21\n", executeProgram("var x : int = 3 * (1 + 9 / 3) - (-
9)\n" +
                "print(x)"));
        assertEquals("21A\n", executeProgram("var x : string = 3 * (1 + 9 / 3) -
(- 9) + \"A\"\n" +
                "print(x)"));
        assertEquals("A21\n", executeProgram("var x : string = \"A\" + (3 * (1 + 9
/ 3) - (- 9))\n" +
                "print(x)"));
    }
```

My contribution to this project was implementing the main funcitonalities of the code base. This included creating the code for the Catscript parser, as well as creating the code for verifying the types of variables and functions compared to their given data or returned data, and generating errors where the program finds them. I also implemented the code for taking inputted Catscript and evaluating the data into usable java data values. Finally I created the code for compiling Catscript into Bytecode for the JVM to evaluate.

Section 3: Design pattern

A design pattern that I implemented within this project is the memoization pattern. The memoization pattern involves creating a hashmap of values, then when calling a function to produce a value, first checking if the value already exists in the hashmap to return it, or creating a new value if the hashmap doesn't have it already and saving it to the hashmap. I included this pattern in the getListType function within the CatscriptType.java class. The pattern is highlighted yellow in the picture below.

I used the memoization pattern rather than just coding it directly to optimize getting list types. If I had just coded it directly, then the program would need to generate a new list type everytime a new list is created. With the memoization pattern implemented, each created list type gets saved in a map, this optimizes my code because it allows the code to look up if a list type of the input Catscript type already exists. It can then return the already created object if it does exist, or if it doesn't already exist it makes a new list type and saves it to the map for future reference.

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

Catscript documentation:

Introduction

Catscript is a simple scripting language that supports primitives data types (Int and Bool) as well as reference types (String, Object, List). Catscript supports primitive for loops, user-defined functions, logic control and a left-associative mathematical order of operations. Catscript does not need (nor does it support) using semicolons to end lines and all whitespace is ignored so any newlines and tabs are solely for the readability of the code. A small example can be seen below, followed by further documentation of the features of Catscript.

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

Expressions

Catscript is a Left Associative language, meaning that operations are completed in order of mathematical precedence from left to right. The only exception is with unary operations which are evaluated from right to left.

Dynamic Typing

Type is determined by the first expression assigned to it. Once this is determined any other type of expression cannot be assigned to that variable

```
var x = "foo"
x = (any other String only)
```

Explicit Typing

Variables can be given explicit types to ensure the compiler doesn't interpret the expression as the wrong type

```
var x : String = "foo"
```

Data Types

String

Java-style string object

```
var x = "foo"
```

Int

This is used for instantiating a variable as a primitive 32 bit integer

```
var x = 32
```

Bool

This is a primitive boolean value, represented as True/False

```
var x = true
```

List

A collection of objects. All objects within the list must be of the same type or the list must be explicitly cast as a list of Objects with every value in the list being assignable to Object. All lists are immutable with a syntax of [e1, e2, e3, ...].

```
var x = [0,1,2]
```

Null

A value that can be cast into any other reference value (String, List, Object), useful for instantiating variables that don't yet have an explicit value

```
var x = null
```

Object

Java-style Object type that can take integers, strings, lists or booleans

```
var x : Object = [0,1,2]
```

Additive Expressions

Additive Expressions use the '+' and '-' operators to add or subtract two expressions. Additive Expressions are used to either add or subtract two numbers or to concatenate one or more Strings with one or more Objects.

```
var x : string = 1 + "one"
var x : int = 1 + 2 - 3
```

Comparison Expression

Comparison Expressions use '>', '>=', '<', '<=' to compare if two integers are Greater Than, Greater Than or Equal, Less Than or Less Than or Equal respectively.

```
var x : Bool = 1 > 2
```

Equality Expression

Equality Expressions use '==' and '!=' to determine if two expressions are equal or not equal respectively. This can be used to compare Ints, Strings, Objects and null expressions.

```
var x : bool = 1 != null
```

Factor Expressions

Factor Expressions use the '*' and '/' operators to multiply or divide two expressions. Factor Expressions can only be used to multipy or divide two expressions of type Int. Due to Factor Expression's precedence over Addition Expressions, Factor Expressions can be included in String assignments so long as no String is multiplied or divided.

```
var x : int = 1 * 2 / 3
var x : String = 1 * 2 + "foo"
```

Function Call Expression

Function Call Expressions are expressions that invoke a function with specified values or variables, if any.

```
var x = foo(a, b, c, ...)
```

Identifier Expression

Identifier Expressions are names of functions or variables. These expressions are used to store or read the value of the variable the same name, or in conjunction with parentheses and the required arguments to call a function.

```
var x = 1
var y = x
var z = foo(a, b, c, ...)
```

Parenthesized Expression

Parenthesized Expressions have the highest precedence, meaning the expression contained within is evaluated first. This is useful for ensuring mathematical operations occur in the desired order.

```
var x = (1 + 2) * 3
```

Unary Expression

Unary Expressions are invert Ints and Bools using '-' or 'not' respectively

```
var x : Int = - 1
var x : String = not false
```

Statements

Catscript relies on a tokenizer to distinguish between lines of code. This means lines do not end with semicolons and that newlines and tabs only impact readibility of the code, not functionality.

Assignment Statements

Assignment Statements evaluate an expression to assign a value to a variable. These statements do not instantiate a variable

```
x = 1
```

For Statements

This loop iterates over a supplied list where each value from the list can be accessed through an identifier

```
for ( IDENTIFIER in [x, y, z]) {

}
```

Function Calls

Functions can be called from anywhere in the program, provided they are called with the necessary arguments. These function the same as a Function Call Expression.

```
functionName1(a, b, c, ...)
var x = functionName2(a, b, c, ...)
```

Function Declarations

Catscript functions are collections of lines of codes that perform specific jobs. These functions can take in variables and either manipulate these variables or return a value. Non-Void functions require a Return statement for every potential branch in the function unless a Return Statement exists as the final line of the function. Typecasting is required for all parameters but functions default to returning an Object type if none is specified. Functions can be declared anywhere within the program and will still be accessible from anywhere within the program.

```
func funcitonName(variable1 : TYPE, variable2 : TYPE, ...) {

}
```

OR when specifying which datatype this function returns:

```
func functionName(variable1 : TYPE, variable2 : TYPE, ...) : TYPE {
    return (VALUE or VARIABLE)
}
```

If Statements

If Statements evaluate a Comparison Expression or an Equality Expression to determine which branch of code is executed. If Statements must always start with an 'if' statement, can have 0 or more 'else if' statements and 0 or 1 else statements.

```
var x = null
if (expression) {
    x = 1
} else if (expression) {
    x = 2
} else if (expression) {
    x = 3
} ...
else {
    x = 0
}
```

Print Statements

Allows the program to print any object to the console

```
print(e1)
print(e1 + "foo")
print("foo" + e1)
```

Return Statements

Return Statements exit the currently invoked function. These can contain a value if an expression is available. A Return Statement must be in any terminable branch of a function, either at the very end of the function or within each branch of the function

```
function foo() : Int{
    if (expression) {
        return 1
    } else if (expression) [
        return 2
    }
    return 3
}
```

Variable Statement

Variable Statements instantiate a variable. These can be explicity typecast or can infer the type from the value of the expression first associated to it. Once a variable is typecast it can no longer take on a value that is not

assignable to it, such as an Int variable can not change to a String variable but an Object variable can be either an Int, String or Bool variable.

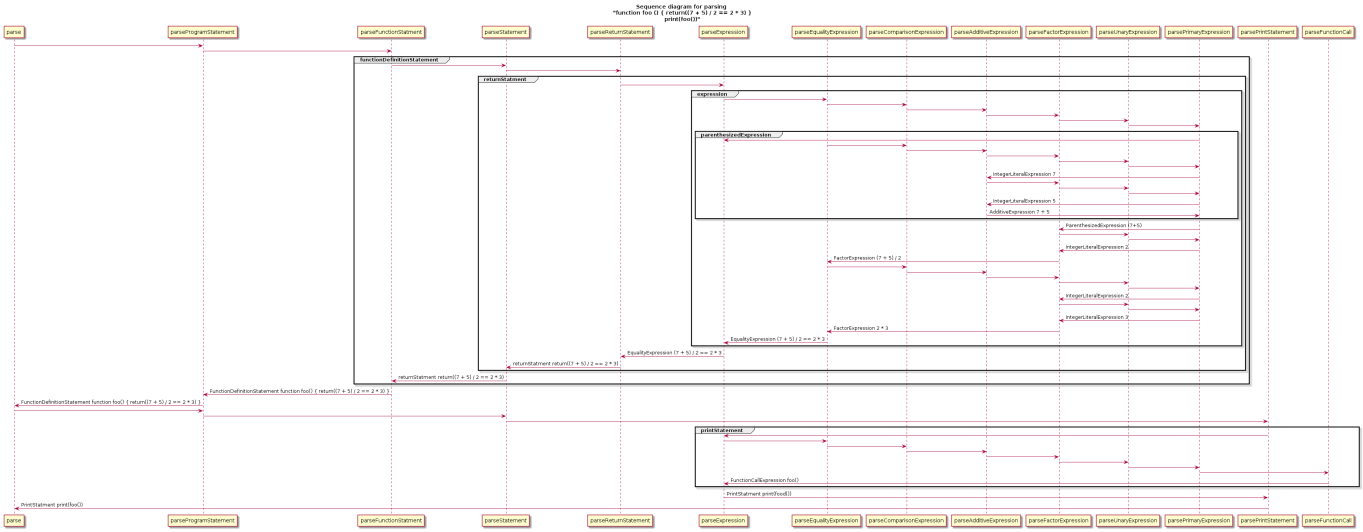
```
var x : Object = "foo"
x = 1
x = true
var y : Int = 1
```

Section 5: UML.

The following is a sequance diagram of the function calls within catscriptParser.java when parsing out the following input:

```
function foo() {
    return((7 + 5) / 2 == 2 * 3)
}
print(foo())
```

This diagram starts in the function parse() and follows all of the recursive descent algorithm into all of the function calls that are executed on the input text. The first group in the diagram outlines the function definition, which goes into the return statement and follows recursive descent until the entire equality expression within the return statement is parsed out. The second group follows the print statement, which goes into the parseFunctionCall function to parse out the function call within the print.



The source image for this diagram can be found in csci-468-spring2024-private/capstone/portfolio/parseUML.png

Section 6: Design trade-offs

For this section, I will discuss the design trade-offs in regards to using the recursive descent algorithm rather than using parser generators.

The recursive descent algorithm is an algorithm that calls parsing functions recursively on an input string to parse the string based on the tokens that you have found. This allows the program to call specific functions to parse out specific statements as the tokens are being consumed, which in turn makes it a lot easier to see the recursive nature of lexical grammars within the code that the developer is creating. This algorithm also allows the developer to hand write the parsing functions, which not only makes the code more readable, but also allows the developer to get a lot more accustomed to and in tune with how their program is reading in statements.

On the other hand, parser generators are tools that allow you to define your lexical grammar rules, then uses the defined rules to generate a parser for you. The upside to using this is that the developer has to write a lot less code by hand to get a parser that is functional. This can make the task of making a parser a lot more trivial for the developer as they can just have a program create everything for them. Using parser generators ensures that the development process will have much less overhead, and allows the developer to create a compiler without having to know a whole lot about the code they are working with.

Over all, I think that using the recursive descent algorithm was a much better design choice for us. The recursive descent algorithm allows developers to have a much deeper understanding of how the grammar that they created for their compiler is working. While using parser generators can be convenient in the way of saving time, the code generated by a parser generator is much harder to follow along with than the code developed for recursive descent. Code generated by parser generators is fairly abstract. They use a lot of hard to read regular expression code along with confusing switch case and logic statements in order to parse out different kinds statements. Especially in the context of our Test Driven Development, stepping through the code that we made makes a lot more sense than having to try and step through a bunch of crazy barely readable regular expression code that a computer made for us. Another plus side to using recursive descent is that it greatly improved our ability to write Java code. Having to write everything out ourselves ensured that we were closely working with the compiler we were building, and also was a very good exercise in sharpening our Java skills as it forced us to write every parsing function by hand and deal with Catscript in the context of how it was being interpreted in Java.

Section 7: Software development life cycle model

The development life cycle we used for this project was Test Driven Development (TTD).

This life cycle I feel is an over all positive. It allows for the developers to have simple tests cases paired with expected inputs, which is very helpful to get a general idea of what the code should be doing. This is really helpful in cases like this project where we are working on a relatively large code base that has already been built, and allows us to make better sense of how we are supposed to be handling the implementation of extra functionality.

I do belive however that this cycle also has some drawbacks. For as helpful as the tests can be, they do not cover all test cases, and they also can be very limiting if the tests are very basic. This came to light when my partner and I started creating tests for each other. We both realized that there were many intended functionalities that were not tested on within the graded test bases, this is especially true of the scoping features in particular. For something that is so important to how the code should be working, the tests included in the assignment tests only test on very rudimentary cases.

The more complex tests that my partner and I created poked a lot of holes in the code base and uncovered a lot of bugs that I feel should have been tested in the main test base. This includes the scoping properties, but also includes many parsing issues that I found as well. A lot of the expression parsing tests only included basic test inputs. Upon running more complex tests, like variable type verifying when the expression is something like a comparison expression or an equality expression should probably have been tested in the original test base, but somehow flew under the radar until after all the premade test base was passing.

Over all, I would say that I do enjoy TDD because it is a very user-friendly way of having people work on large code bases that they did not create. It allows developers to see the end product of what the original developer intended for the functionality, and lets the current developer find their own way to approach getting the test input to output what the original developer intended.