Section 1: Program.

I have attached the source zip file inside my /capstone/portfolio directory.

Section 2: Teamwork.

In this capstone project, each of us created 3 tests for Catscript and we exchange the 3 tests. I need to make sure that the 3 tests that my partner created pass in my compilers and the other way around in which my partner need to make sure that the test I created pass in his compilers. The tests we created is some complex function like a nested if statement, print inside a function, and print a variable expression. This is to make sure that even though the tests that we created consist of different statements or expression, it should still work. Another thing that I did with my partner is that we exchange is the Catscript documentation. Each of us will be writing a documentation of Catscript that specifies all the functionalities and features of Catscript. This documentation must consist of all the features of Catscript such as all the complete expression and statements and include the description and examples for each expression and statements. This documentation will also include an introduction and is divided into several sections. For example, like expressions header have different types of expression which can be formatted using a sub header inside the expression header. With this, it would make the reader easier to read and understand the documentation. This teamwork can be described as I am the primary engineer and my partner is the documentation and testing engineer.

Section 3: Design pattern.

Below is the code in my capstone project where I implement memoization design pattern.

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new
HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);
    if (listType != null){
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return listType;
}
```

One design pattern that is used in my capstone project is Memoization. This design pattern is implemented by store the thing somewhere and if someone calls this method with the same argument, look it up it that storage system rather than creating a new one. Above is the code that I use in my capstone project which implement this memoization design pattern. This above code shows that if I call that with int for the first time, it looks into the map and there is nothing there, it returns null. This then create a new ListType wrapping the int type so it becomes a list of int and putting it into the map and then return that. However, when the second time it calls int again, it looks at the map and there is already int there and so it doesn't re-initialize that and return the original list type. This code is implementing the memoization design pattern. By doing this memoization design pattern in my capstone project, it saves the computation of newing up a list type when it gets called with the same type over and over again.

Section 4: Technical writing.

As we can see below is the below is the Catscript documentation.

Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

Catscript is a simple scripting language. Here is an example:

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

Comments:

Comments are handled with "//" on the line at the point you want to comment:

```
//this is a comment print(12)
//the above print will not run
//but this one will
print(44) //since the comment is after the statement
```

Features

For loops:

For loops give the ability to loop over code multiple times even if the number of iterations changes each run. A for loop in Catscript looks like this:

```
for( x in [1, 2, 3, 4] ) { //the variable x can be used later on in the loop as it is a local variable
print(x)        //prints: 1 2 3 4
}
```

To use a for loop, you must provide a list to iterate over. There can be as many other statements including other for loops inside a loop.:

```
for( x in [1, 2, 3, 4] ) {      //outside loop
    print(x)    //prints: 1 2 3 4 each time the loop is run
    for( y in [1, 2, 3, 4] ) { //nested loop
        print(y)        //prints 1 2 3 4 each time the loop runs
    }
}
OUTPUT: 1 1 2 3 4 2 1 2 3 4 3 1 2 3 4 4 1 2 3 4
```

If statements:

If statements are conditionals for testing if a condition contained inside the statement is true. For example:

```
if(true){
    print("True")
}else{
    if(false){
        print("nested  False")
    }else{
        print("nested  True")
    }
}
```

If the first condition is true, then the first print: "Print("True")" will execute/run. If the first condition is not true, then the else runs. If statements are also "nest-able" meaning that they can be inside themselves as shown above in the example.

Variable assignment:

Variables can be assigned with an explicit type or with a implicit type. For example:

```
var x = "Hello World!"  //declare variable with implicit typing
var y : int = 54        //declare variable with explict typing

print(x)        //prints: Hello World!
print(y)        //prints: 54

x = "Goodbye!"  //redeclare a variable after assignment/creation
print(x)        //prints: Goodbye!
```

Types:

The supported types are as follows:

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

If an explicit type is not found from evaluation or declaration (var x : int) the type is object

Lists

Lists are supported in Catscript and are declared as follows:

```
var lst :list<int> = [1, 2, 3, 4]       //declare a list of type integer

for( x in lst) {        //iterate over the list
    print(x)    //prints: 1 2 3 4
}
```

List types supported are all basic types: int, string, boolean, list, and object.

Functions:

Functions can be created and called like so:

```
function x (){   //declare a function with the keyword "function" then the function name
    var q = 4    //create variable
    if(q > 3){   //check if condition is true
        print(q) //prints: 4
    }else{
        print("q is not greater than 3!")
    }
}

x()  //fucntion call
```

Functions can have as many statement's and expressions as desired in them, as shown above.

Functions can also have parameters passed into them:

```
function x (y, z: int){   //multiple parameters can be passed in with commas between them
    print(y)   //prints: String
    print(z)   //prints: 1
}

x("String", 1) //calls x with y = (String) and z = (1)
```

As shown above, you can also have the type of the parameters be implicit or explicit as desired. There is no limit to how many parameters are passed into a function, nor a minimum.

Argument lists:

Functions can have an argument list passed in when invoked to fill the required parameter list. Example:

```
function f(x:int, y:string){
    print(y)        //prints y
    print(x + 3)    //prints the result of x + 3
}

f(3, "Hello")   //prints Hello 6
```

These argument lists are required to be as long as the parameter list for the function definition is and if the type is not specified it is treated as a string.

Built-in Function(s):

Catscript currently only has one built-in function, that is the print function that is being used in many of the examples. It only supports printing a single expression at the moment, for example:

```
print(1)              //prints: 1
print(1 + 1)          //prints: 2
print(10 / 5)         //prints: 2
print("Hello World!") //prints: Hello World!
```

Returns:

A function can return a value from it and hold it in a variable or use it for a calculation. An example of returns are as follows:

```
function x(){
    return 1  //returns the int value 1
}
print(x())   //uses the returned value to print 1
var y = x() + 2   //uses the return value to perform a math calculation and store it in a variable
print(y)          //prints out the value of the variable using the returned value
```

As shown above, a function can be called and the return value used for other purposes. The above code uses the return value from x() returning 1 to do a math calculation. If a return is placed outside a function definition it is treated as a syntax error.

Expressions:

Expressions are evaluated in order of (left to right) Primary>Unary>Factor>Additive>Comparison>Equality (with left being evaluated first and right being evaluated last, this is also the way binding strength scales with left being strongest and right being weakest).

Parenthesized expressions:

Wrapping expressions in a parentheses is supported in Catscript, this can allow for more clearly showing what order expressions should be evaluated in, for example:

```
print(2 * (2 + 4)) //result is 12
print(2 * 2 + 4)   //result is 8
```

Primary Expressions:

Primary expressions are an expression that evaluated to a basic type, a variable, a function call, a list, or a parenthesized expression. The supported types are: true, false, null, list_literal, function_call, (expression), Identifier, String, Integer.

Unary Expressions:

A unary Expression gives the ability to negate a primary expression, example:

```
if(not true){   //use not to negate a boolean
    print("Not True")   //not reached
}
if(-1 < 0){     //negates a number
    print("-1 is less than 0")  //prints: -1 is less than 0
}
```

The '-' does not work with Boolean's and the "not" keyword is not for negating numbers.

Factor Expressions:

A factor expression supports multiplication and division between numeric unary expressions. Example:

```
print(10 / 5)    //prints: 2
print(10 * 5)    //prints: 50
print((2 + 4) * (2 / 2))         //prints: 6
```

As shown above, nesting expressions does not matter as the unary is evaluated for each side of the sign first.

Additive Expressions:

An additive expression handles both concatenation of strings and addition of numbers, Example:

```
print(5 + "hello")       //prints: 5hello
print(4 + 5)     //prints: 9
print( 5 - 5)    //prints: 0
```

Comparison and Equality Expressions:

All equality and comparison expressions are left to right comparatively, meaning that: left > right, checks if left is Greater than right.

Comparison expressions are treated as more tightly binding than equality expressions.

Comparison Expressions:

Catscript supports 4 comparison expressions: < : Less >: Greater <=: Less or Equal >=: Greater or Equal Below are examples of each:

```
if(2 > 1){       //checks if 2 is greater than 1
    print("Greater!")    //prints: Greater!
}
if(4 >= 4){      //check if 4 is greater or equal to 4
    print("Greater or Equal!")  //prints: Greater or Equal!
}
if(1 < 4){       //checks if 1 is less than 4
    print("Less!")       //prints: Less!
}
if(3 <= 4){      //checks if 3 is less than or equal to 4
    print("Less or Equal!")      //prints: Less or Equal!
}
```

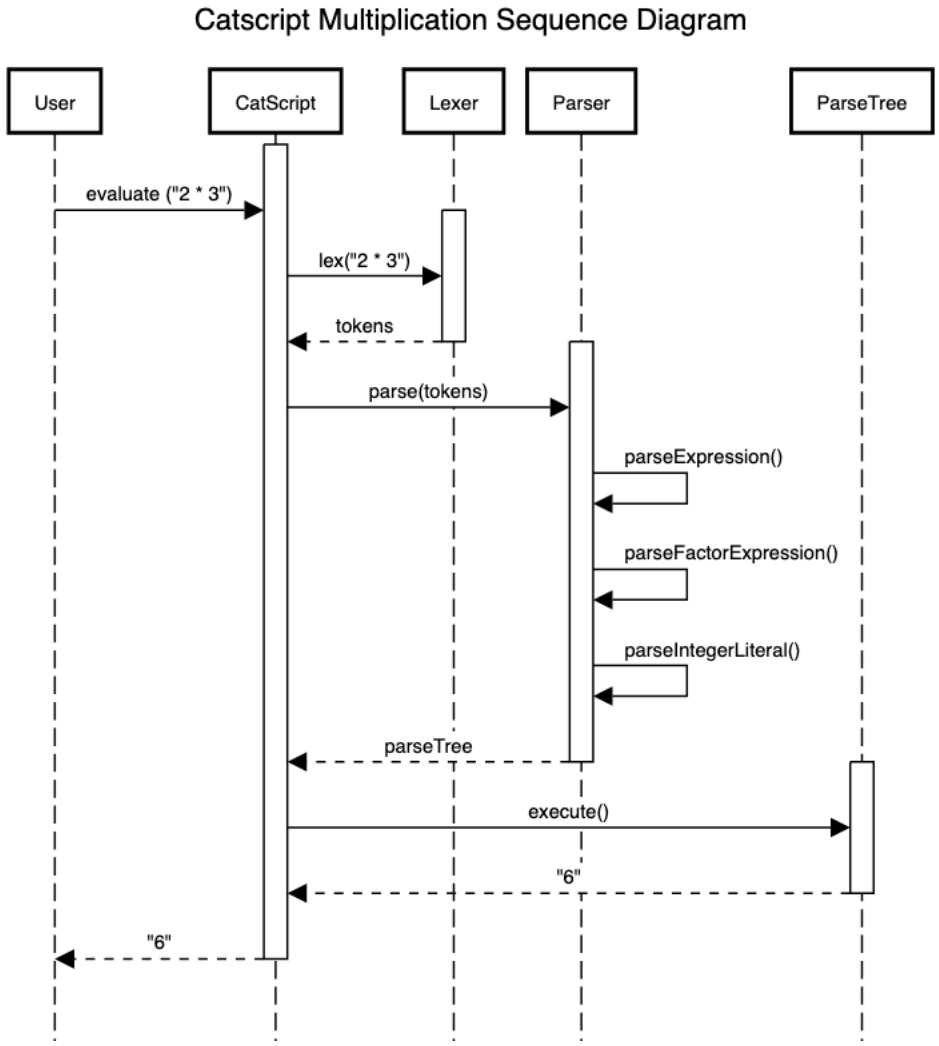
Equality Expressions:

Catscript supports 2 equality expressions: ==: Equal !=: Not Equal Below are examples of each:

```
if( 1 == 1){    //checks if 1 is equal to 1
    print("Equal!")     //prints: Equal!
}
if(1 != 5){     //checks if 1 does not equal 5
    print("Not Equal!") //prints: Not Equal!
}
```

Section 5: UML.

Below is the UML Sequence Diagram for the steps of multiplication in Catscript.



Section 6: Design trade-offs.

The design trade off decision that I made is doing parser by hand which uses a recursive descent algorithm instead of using parser generators. Most of the parser in other universities uses parser generators when creating parsers and I've notice that writing a parser by hand can be a lot easier, simpler, and nicer to create parser than using a parser generator. Parser generators take a language specification and generate a parser for that specification. The input will most likely be a lexical grammar in some sort of Regular Expression and a language grammar in some sort of EBNF. This will generate a lexer or parser for you and the code will be very hard to be read which makes it hard to debug aswell. In addition, the code that it will generate will be way longer than if we were writing parser by hand. On the other hand, this recursive descent algorithm is widely used in industry and it teaches the recursive nature of grammars in a very clear way. This algorithm is way more flexible since we can create our own execute method to execute something and we could modify it. The code is also more readable because we are doing it by hand, and which makes it easier to debug. Moreover, the code that we wrote using this recursive descent algorithm is also way shorter and simpler which is nicer for creating a parser. Therefore, I decided to use the recursive descent algorithm in my capstone project compared to using parser generators.

Section 7: Software development life cycle model.

The model that I used to develop my capstone project is using a Test Driven Development (TDD). There is a bunch of test suite and is divided into different sections (tokenizer, parser, eval and bytecode). This test suite is to check and make sure that all the expression or statements is doing the right thing. If I pass all the test, this means that it does the right thing that it's supposed to do. I need to pass all those test suite in this capstone project and this test helps me check if there's anything wrong with my compiler. This test model is very useful to develop my capstone project as I know which one is not working and fix it from there.