

# **Senior Capstone Portfolio**

**Compilers 468**

**Spring 2023**

**Team Members:**

**Joseph DiPrizio**

**Tyler Koon**

# **Capstone Documentation**

**Joseph DiPrizio**

**May 2023**

## **Section 1: Program**

Section one of the capstone project is the code that is implementing a compiler for the Catscript programming language. To fulfill this section, I have included a zip file with this document that contains all of the code that allows the Catscript compiler to function, as well as all of the tests that were used to guide the development of the Catscript compiler.

## **Section 2: Teamwork**

I worked with Tyler Koon, who I will refer to as team member one on this project. Team member one focused on the Catscript documentation for the compiler, and I worked on the implementation and coding of the compiler. The documentation goes in depth into the different features that Catscript provides and gives examples of how to use these features. This technical document can be found in section four of this documentation. Member one also wrote several tests that are contained in the CatscriptCapstoneTests.java file in the zipped code file. These tests ensured that the compiler is able to handle if statements that utilize a return statement, for loops that are nested within one another, and for loops combined with several features that modify control flow, such as if statements or function calls. The Java code for the tests is the following:

```
package edu.montana.csci.csci468.eval;
import edu.montana.csci.csci468.CatscriptTestBase;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
public class CatscriptCapstoneTests extends CatscriptTestBase {
    /**
     * The following test verifies that 'if' statements handle return behavior correctly.
     * That is, if they encounter a return statement during execution, they do not run any following
code.
     */
    @Test
    void ifStatementHandlesReturn() {
        // The following program should print 5 and not -1
        assertEquals("5\n", executeProgram(
            "var x = 0" +
            "function validReturn() {" +
            "    if(x == 0) {" +
            "        x = 5" +
```

```

        "return" +
        "}" +
        "x = x - 1" +
        "}" +
        "validReturn()" +
        "print(x)")
    );

```

// The following program should print 10 and not 0  
 assertEquals("10\n", executeProgram(

```

        "var x = 1" +
        "function validReturn() {" +
        "    if(x == 0) {" +
        "        x = 5" +
        "        return" +
        "    } else {" +
        "        x = 10" +
        "        return" +
        "    }" +
        "    x = x - 1" +
        "}" +
        "validReturn()" +
        "print(x)")
    );
}
/**

```

\* The following test verifies that 'for' statements can be nested, and iterate the parent iterables in the

\* correct order.

\*/

@Test

```

void forStatementsCanBeNested() {
    assertEquals("1\n4\n5\n6\n2\n4\n5\n6\n3\n4\n5\n6\n", executeProgram(
        "var x = [1, 2, 3]" +
        "var y = [4, 5, 6]" +
        "for(i in x) {" +
        "    print(i)" +
        "    for(j in y) {" +
        "        print(j)" +
        "    }" +
        "}"
    ));
}

```

/\*\*  
 \* The following test verifies that 'for' statements can include compound statements, such as function calls and if

\* control flow logic like 'if' statements.

\*/

@Test

```

void compoundForStatements() {

```

```

assertEquals("1\n2\n3\n", executeProgram(
    "var x = [1, 2, 3]" +
    "function test(y: int) {" +
    "    print(y) +
    "}" +
    "for(i in x) {" +
    "    test(i) +
    "}"
)
);
assertEquals("2\n", executeProgram(
    "var x = [1, 2, 3]" +
    "for(i in x) {" +
    "    if(i == 2) {" +
    "        print(i) +
    "    }" +
    "}"
)
);
assertEquals("-1\n2\n-1\n", executeProgram(
    "var x = [1, 2, 3]" +
    "for(i in x) {" +
    "    if(i == 2) {" +
    "        print(i) +
    "    } else {" +
    "        print(-1) +
    "    }" +
    "}"
)
);
}
}

```

I primarily worked on the code for the Catscript compiler. I started with tokenization, which involved analyzing input and grouping it into different categories. For example, input such as 123 is tokenized to be an integer, and x was tokenized as an identifier. Parsing was the next step, which was broken into parsing expressions and parsing statements. The difference is expressions evaluate to a value, such as "1+1", whereas statements act as a sort of action, like "print(x)" or a for loop. Once the compiler was able to tokenize and parse input, the last steps were evaluation and JVM bytecode. This step involved going through each type of statement and expression and breaking it down into bytecode so the compiler was able to evaluate it, and we could compile Catscript code down to JVM bytecode.

Concerning time spent on the project, team member one spent approximately six to eight hours on the technical documents and tests for the Catscript compiler. I spent approximately 40-50 hours implementing the compiler. This time was spent doing the following:

Tokenization	6 hours
Expression Parsing	6 hours
Statement Parsing and Evaluation	18 hours
Compilation	12 hours

### **Section 3: Design Pattern**

For the design pattern, I chose to memoize the method `getListType()` that can be found in the `CatscriptType.java` file. By memoizing this call, we store a `ListType` into a hashmap if we call the function and it is the first time that `ListType` has been seen by the function. This means that whenever we call `getListType()` again attempting to get the same type, we can retrieve it from the Hashmap instead, thus saving resources. This code has been highlighted in the `CatScriptType.java` file, and an image of the memoized function has been included at the end of this section.

```
35 //MEMOIZED CALL
36 //*****
37 private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<CatscriptType, CatscriptType>();
38 public static CatscriptType getListType(CatscriptType type) {
39     //return new ListType(type);
40     CatscriptType listType = LIST_TYPES.get(type);
41     if(listType == null) {
42         listType = new ListType(type);
43         LIST_TYPES.put(type, listType);
44     }
45     return listType;
46 }
47 //*****
48
```

### **Section 4: Technical Writing**

The technical writing document was written by team member one, as was mentioned earlier in the teamwork section. I have appended it onto the end of this page.

# CatScript Documentation

Author: Tyler Koon

- CatScript Documentation
  - Introduction
  - Type System
    - \* Concrete Types
    - \* List Types
  - Control Flow
    - \* For Loops
    - \* If Statements
    - \* Functions and Return Statements
  - Assignment
    - \* Identifiers
    - \* Variables
  - Lists
    - \* Defining Lists
    - \* Accessing Lists
  - Operators
    - \* Arithmetic Operations
    - \* Logical Operations

## Introduction

CatScript is a high-level, C-like programming language that has been developed for educational purposes. Though a small and simple language, it supports a number of modern features, including a static type system, control structures such as loops and functions, variable assignment, lists, and basic algebraic and comparative operators. This documentation covers these features in detail, providing plenty of examples that demonstrate the CatScript language in action.

## Type System

CatScript implements a simple static type system that uses a syntax similar to TypeScript. Using a static type system promotes a more rigid development process, and allows the CatScript compiler to provide more comprehensive error messages that supports a more convenient development environment.

Here is what it might look like to explicitly assign a variable as an integer:

```
var myNum: int = 10
```

Of course, CatScript also supports type inference. The following `myNumPlusFive` variable will also be identified as an integer:

```
var myNumPlusFive = myNum + 5
```

## Concrete Types

In CatScript, there are six concrete types: *Integers*, *Strings*, *Booleans*, *Objects*, *Null*, and *Void*. These types and their Java mappings are provided in the table below.

Type	Name	Java Mapping	Description
int	Integer	<code>java.lang.Integer</code>	32-Bit Integer
string	String	<code>java.lang.String</code>	String Value
bool	Boolean	<code>java.lang.Boolean</code>	Boolean Value
object	Object	<code>java.lang.Object</code>	Any Value
null	Null	<code>java.lang.Object</code>	Null Type
void	Void	<code>java.lang.Object</code>	Void Type

These concrete types follow a simple set of assignability rules. The *int*, *string*, and *bool* types are all assignable to the *object* type. For example, an Integer-type variable can be assigned to an Object:

```
var myInt: int = 10
var myObject: object = myInt
```

The same can be done with a String:

```
var myString: string = "Lorem"
var myObject: object = myInt
```

And a Boolean:

```
var myBool: bool = true
var myObject: object = myInt
```

That said, the Object types *cannot* be assigned to its primitive children (Integers, Strings, and Booleans). For example, you are not allowed to assign a variable of type *object* to another variable of type *int*:

```
var myObject: object = 10
```

```
// This is not allowed
var myInt: int = myObject
```

As a general rule of thumb, the assignability of the Integer, String, Boolean, and Object types match the assignability rules of the corresponding Java types.

The Null and Void types are special in the CatScript language. Much like in other high-level languages, the Null type simply corresponds with the `null` value. Every type is assignable from the null type. For example, you can initialize a variable as null, and then assign it to an integer later on:

```
// `futureInt` initialized to null; inferred type is `null`
var futureInt = null

// ... do something ... //
```

// The `futureInt` variable can then be assigned to an integer; inferred type is `int`

```
var futureInt = 10
```

On the other hand, nothing can be assigned from the void type. Thus, the following assignment would not be allowed

```
// This is not allowed
var nonSense: void = 10
```

Instead, the void type is only used to identify functions that have no return value:

```
// This is allowed
function myFunc(): void {
    print("I do not return anything")
}
```

## List Types

In addition to the six concrete types, CatScript implements a complex List type that supports parameterization. The syntax for this type is `list<T>`, where `T` is the component type for the List object. For example, assigning a variable to a list of integers would look like:

```
var myList: list<int> = [1, 2, 3]
```

The component type supports all six concrete types in addition to the list type. This allows for the assignment of nested lists, such as a list of lists of Integers:

```
var myNestedList: list<list<int>> = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Much like the Array type in Java, the CatScript List type is *covariant*; that is, a list `L1` is assignable to list `L2` if, and only if, the component type of list `L1` is assignable to the component type of list `L2`. Unlike Java arrays however, **lists are immutable** which prevents logical assignment errors, such as inserting an Integer into a list of Objects.

To demonstrate the covariance of lists, consider assigning a list of strings to a list of objects:

```
var myStringList: list<string> = ["a", "b", "c"]
```

```
var myObjectList: list<object> = myStringList
```

This would be valid because the `string` type is assignable to the `object` type. Attempting to assign a list of objects to a list of strings however would be invalid:



```

var myObjectList: list<object> = ["a", "b", "c"]

// This is not allowed because type 'object' is not assignable to type 'string'
var myStringList: list<string> = myObjectList

```

## Control Flow

In CatScript, there are three basic control structures: for loops, if statements, and functions. These allow for advanced programming features, including iteration, abstraction, and complex conditional logic.

### For Loops

As with most other languages, the *for loop* allows for iterations through iterable items. Similarly to these other languages, the for statement takes on the following form:

```

for(IDENTIFIER in ITERABLE_EXPRESSION) {
    STATEMENTS
}

```

The IDENTIFIER will infer its type from the ITERABLE\_EXPRESSION. So for example, when iterating through a list of integers, the IDENTIFIER will be of type `int`. In CatScript, List are the only valid ITERABLE\_EXPRESSION. Iterating through a list of integers might look like:

```

var myInts: list<int> = [1, 2, 3]

// myInt infers the type of 'int' from myInts
for(myInt in myInts) {
    // Do something
}

```

The body of the for loop can consist of any number of valid CatScript statements, including additional *for loops* and the Return statement (which will cause the loop to terminate prematurely). As an example, consider the following fully qualified for loop that searches for a string in a list of strings:

```

var words: list<string> = ["cat", "script", "catscript"]

// word infers the type of 'string' from words
for(word in words) {
    // Check if word is equal to my favorite word
    if(word == "catscript") {
        // If so, print a message and terminate the for loop
        print("I have found my favorite word")
        return
    }
}

```

```
    print("I have not found my favorite word")
}
```

## If Statements

Another common control structure implemented in CatScript is the *if statement*. Once again, this structure performs similarly to most other high-level languages, taking the following form:

```
if(EXPRESSION) {
    STATEMENTS
}
```

The **EXPRESSION** can be any valid CatScript expression that evaluates to a truthy value (i.e., true or false), and the **STATEMENTS** can consist of any valid set of statements, including additional *if statements* for compound conditional logic. When the **EXPRESSION** evaluates to true, the **STATEMENTS** will be executed. Otherwise, if the **EXPRESSION** evaluates to false, *none* of the **STATEMENTS** will be executed. An example of this behavior is demonstrated below:

```
if(true) {
    // These statements will be executed!
    print("This will be printed")
}

if(false) {
    // These statements will not be executed!
    print("This will not be printed")
}
```

CatScript also supports **else** statements, allowing for the chaining of conditional logic. The statement can be followed by either an additional *if statement* or a set of **STATEMENTS** that will be executed if the preceding **EXPRESSION** is falsy. As an example, consider this chain of if/else statements that reacts to an integer value:

```
var myInt: int = 10

if(myInt == 5) {
    // These statements will be executed if myInt is equal to 5
    print("My integer is five!")
} else if (myInt == 10) {
    // These statements will be executed if myInt is not equal to five, but is equal to 10
    print("My integer is ten!")
} else {
    // These statements will be executed only if myInt is not equal to 5 or 10
    print("I do not know what my integer is :(")
}
```

## Functions and Return Statements

The final control structures in the CatScript language are functions and return statements. As with most other languages, functions can be used to associate a set of statements with an identifier, which can be called later on in the program. A function declaration simply consists of an *identifier*, a set of *parameters*, and a set of *statements*:

```
function IDENTIFIER(PARAMETERS) {  
    STATEMENTS  
}
```

The IDENTIFIER can be any valid identifier, and is used to call the function from elsewhere in the program. The syntax for calling a function is simply the function's identifier followed by a set of parenthesis containing a comma-delimited list of the functions parameters:

```
var myInt: int = 5  
  
function myFunc(paramOne) {  
    // Do something  
    print(paramOne)  
}  
  
// Calling myFunc if the value of myInt is 5  
if(myInt == 5) {  
    myFunc(myInt)  
}
```

Additionally, the parameters in the PARAMETERS list can be explicitly types. If they are not explicitly type, then the types will be inferred:

```
function add(a: int, b: int) {  
    // Variables a and b have been explicitly typed as integers  
    print(a + b)  
}
```

In the CatScript language, the body of functions execute in their own scope, allowing for the creation of local variables that persist only within a function call. For example, the variables `a` and `b` in the preceding example are only accessible within the body of the `add` function. That said, functions have access to any variables declared in the functions parent scope:

```
var five = 5  
  
function addFive(a: int) {  
    // The variable five can be accessed from within `addFive` because  
    // `five` is declared in the parent scope  
    print(a + five)
```

```
}
```

Finally, function bodies can contain a **return** statement that allows for execution to be halted, and for a value to be returned to a function call.

```
function multiply(a: int, b: int): int {  
    // Return an integer representing the product of a and b  
    return a * b  
}
```

```
var myValue = multiply(a, b)
```

## Assignment

In CatScript, identifiers are to associate human-readable labels that with values. These can be used in operations and control structures to compare and mutate data.

### Identifiers

An identifier is any valid alphanumeric string that is not a CatScript keyword and does not begin with numerals or special characters. Examples of valid and invalid identifiers are provided below:

Valid Identifier	Invalid Identifier
MyIdentifier	@myidentifier
MyIdentifier123	123MyIdentifier
forloop	for

These identifiers can be assigned to any valid CatScript value (including functions) using Variable Statements. Additionally, identifiers can be used in valid Operations, or passed between Control Structures.

### Variables

To assign a value to an Identifier, you can use the *var* statement. This functions similar to its Java counterpart, allowing valid CatScript values to be associated with a label. Assigning the identifier **myNumber** to the integer value 10 would look like:

```
var myNumber = 10;
```

As has been demonstrated throughout this documentation, the variable statement can also accept explicit typing information. Consider the same exmaple of assigning an integer, but this time typing the **myNumber** variables as object:

```
var myNumber: object = 10;
```

Note: In CatScript, a variable statement *must* be assigned to a value at creation; you cannot create uninitialized variables

In addition to the traditional value types described in the Type System section, identifiers can also be assigned to function definitions:

```
var myFunction = function(): int {  
    // Do something  
    return 1  
}
```

which can then be used to invoke a function declaration (the result of which can also be assigned to an identifier):

```
var myFunctionResult = myFunction()
```

## Lists

Lists are one of the more complex offerings of the CatScript language. They allow for multiple instances of data to be stored in a single, iterable collection.

### Defining Lists

A list is defined by a collection of comma delimited values between two brackets ([ and ]). For example, creating a list of integers would look like:

```
var myInts = [1, 2, 3, 4, 5]
```

As previously mentioned, lists can be explicitly typed using the special `list` type, which accepts a component type ( the explicit type of the contents of the list). Explicitly typing the previous list might look like:

```
var myInts: list<int> = [1, 2, 3, 4, 5]
```

Lists can contain any valid CatScript expression (including additional lists), however all elements of the list must be of the same type:

```
// This is allowed  
var myListOfLists: list<list<int>> = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
  
// This is not allowed  
var myListOfLists: list<list<int>> = [[1, 2, 3], ["Some", "string"], false]
```

Additionally, recall from the Type System section that the list type is covariant, allowing for the assignment of lists according to the assignability of a lists component type. Once again, this covariance is safe because *CatScript lists are immutable*; that is, **lists cannot be updated after creation**.

### Accessing Lists

The only way to access lists in CatScript is through iteration, which can be achieved using a For Loop. The for loop populates an intermediate identifier

with each value of a provided list, allowing individual access to all elements in a list. An example of how you might iterate through a list of integers is provided below:

```
var myInts: list<int> = [1, 2, 3]

# Print out each element in the 'myInts' list
for(value in myints) {
    print(value)
}
```

Recall that intermediate identifier ('value' in the above example) infers its type from the provided iterable. For more examples of using iteration to interact with elements in a list, refer to the section on For Loops.

## Operators

Finally, we consider the various operators implemented in the CatScript language. These operators can be classified into two categories: arithmetic and logical. For both of these categories, the CatScript language adopts right-associativity rules. This means that expressions will be evaluated from right to left, unless the order of operations is explicitly defined through the use of parenthesis. As an example, consider the following additive expressions:

```
// This would be evaluated as 1 + (1 + 2)
1 + 1 + 2
```

```
// This would be evaluated as (1 + 1) + 2
(1 + 1) + 2
```

Additionally, all of these operators can accept any valid CatScript expression, however the compiler will fail to parse expressions that are not supported by the operation. For example, you can take the product of two identifiers which represent integer values:

```
var valueOne: int = 2
var valueTwo: int = 2

// This is allowed (returns the integer '4')
var four: int = valueOne * valueTwo
```

However, you can not take the product of two identifiers that do not represent integers:

```
var valueOne: string = "test"
var valueTwo: bool = false

// This is not allowed because valueOne and valueTwo are not valid integers
var four: int = valueOne * valueTwo
```

## Arithmetic Operations

Arithmetic operators consume and return numerical values. In CatScript, there are three such operators: Additive, Factors, and Unary. The *additive expression* supports addition and subtraction operations:

```
// Sum two integers (prints '2')
print(1 + 1)
```

```
// Subtract three integers (prints '1')
print(3 - 1 - 1)
```

The *additive expression* is unique in that it can also support the concatenation of strings by adding two string values:

```
// This is allowed (prints 'Mystring1')
print("My" + "string" + 1)
```

The *factor expression* supports multiplication and division:

```
// Product of two integers (prints 4)
print(2 * 2)
```

```
// Quotient of two integers (prints 2)
print(4 / 2)
```

Note: CatScript only supports integer values, so any fractional result will have the ‘floor’ operation applied

And the *unary expression* supports negation:

```
// Negate an integer (prints '-1')
print(-1)
```

```
// Negate an additive expression (prints -5)
print(-(6 - 1))
```

The *unary expression* is also unique in that it can support both numerical values and logical values. See the Logical Operators section for more details on this behavior.

## Logical Operations

Logical operators consume two expressions and return a boolean value. In CatScript, there are three such operators: equality, comparison, and unary. The *equality expression* can be used to assess whether two expressions are equal, and like most other high-level language uses the == and != syntax:

```
// Check if two integers are equal (prints 'true')
print(1 == 1)
```

```
// Check if two strings are equal (prints 'false')
print("orange" == "apple")
```

```
// Check if two strings are equal (prints 'true')
print("orange" != "apple")
```

This equality operator is expanded upon by the *comparison expression*, which supports the following operations:

Operation	Syntax
Less Than	<
Less Than or Equal	<=
Greater Than	>
Greater Than or Equal	>=

```
// Compare two integers (prints false)
print(5 > 5)
```

```
// Compare two integers (prints true)
print(5 >= 5)
```

```
// Compare two integers (prints true)
print(5 < 6)
```

```
// Compare two integers (prints false)
print(7 <= 6)
```

Finally, the *unary expression* can also be used for logical negation by using the ! (bang) operator:

```
// Negate a boolean value (prints false)
print(!true)
```

```
// Negate a boolean value (prints true)
print(!(1 == 5))
```

## Program Example

As demonstrated by this documentation, the CatScript language is small and simple, offering an elementary approach to programming. That said, don't let this trick you into thinking that CatScript is ineffective at completing complex, multimodal programming tasks. From a static typing system to fully qualified functions and control structures, CatScript is quite a capable language. As an demonstration of this capability, consider the following CatScript program which iterates through two lists of integers and identifies if there is a combination of values between those lists that sum to a target value:



```

// Function to identify if two lists contain a combination of integers that sum to a target
function sumsToTarget(a: list<int>, b: list<int>, target: int) {
    // Iterate through each combination of elements between lists a and b
    for(value in a) {
        for(value in b) {
            // If the two values sum to the target, return true
            if((a + b) == target) {
                return true
            }
        }
    }

    // If no two values sum to the target, return false
    return false
}

// Define lists and targer
var listOne: list<int> = [2, 7, 3, 5]
var listTwo: list<int> = [3, 8, 4, 6]
var target: int = 7

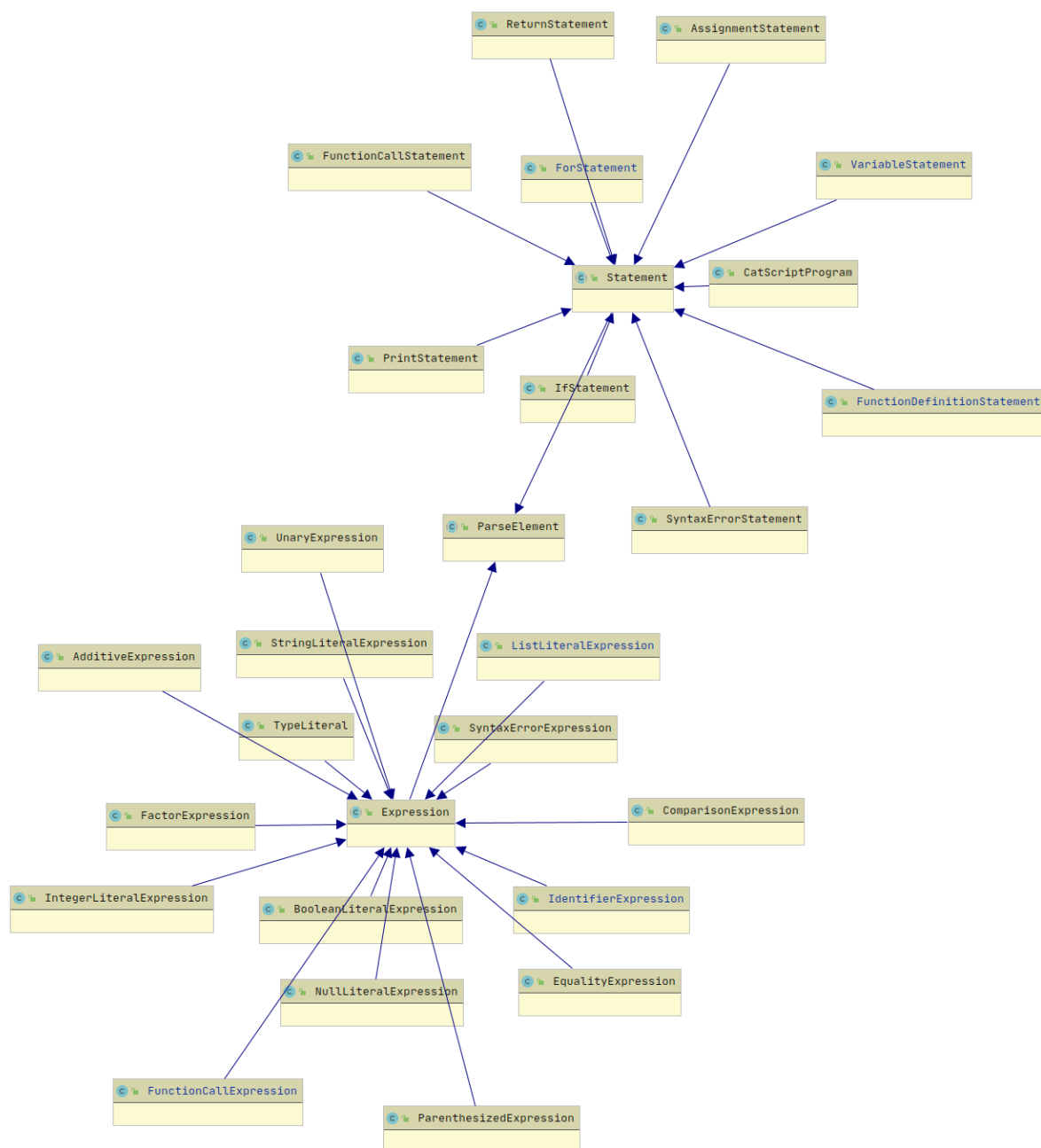
// Check if the lists contain a combination of values that sums to the target
var doesSumToTarget: bool = sumsToTarget(listOne, listTwo, target)

print("Lists contain a combination of values that sum to " + target + ": " + doesSumToTarget)

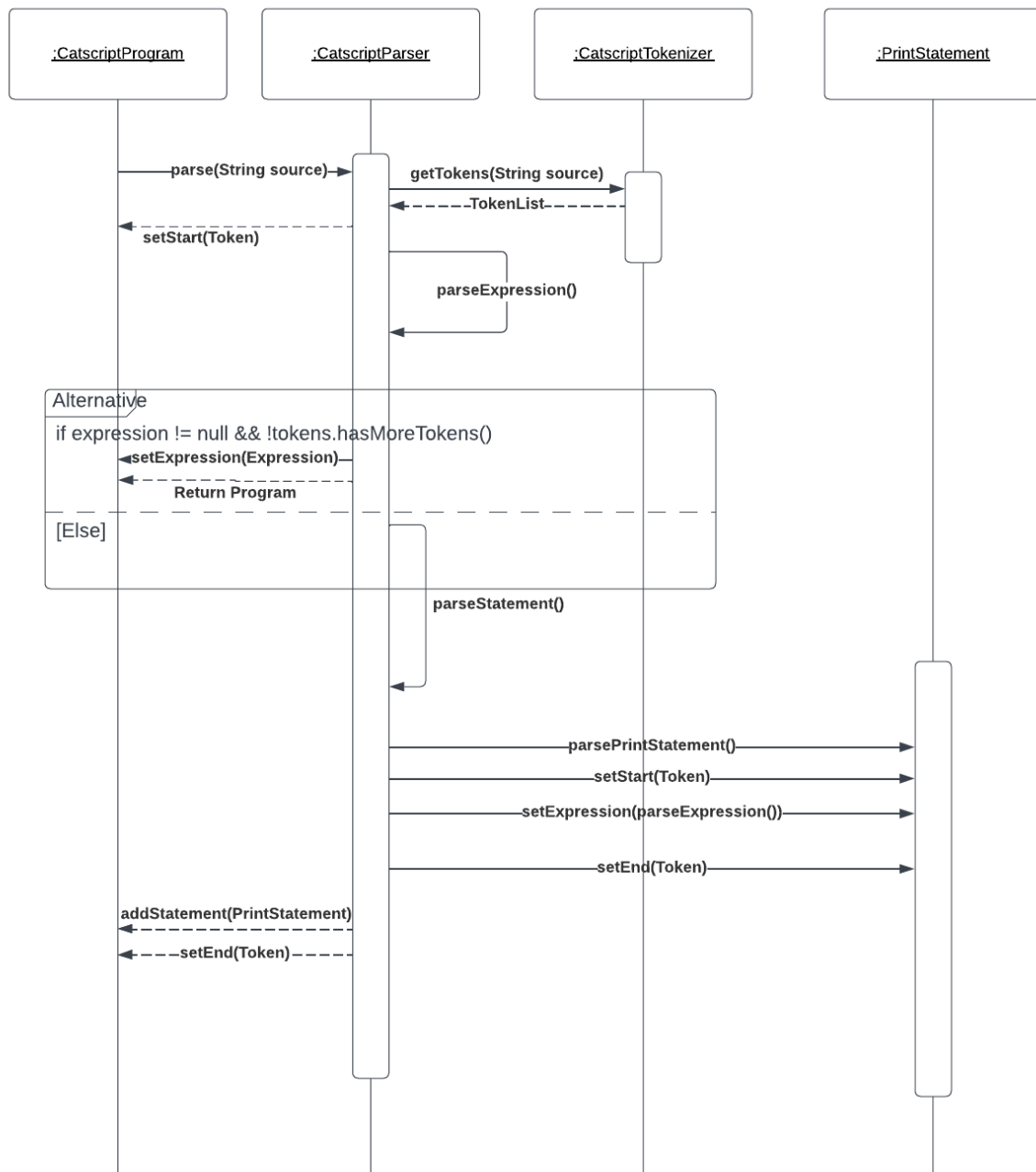
```

## Section 5: UML

Analyzing a few UML diagrams, we first created a diagram that visualizes how we are using statements and expressions. When we parse an element, it either parses a statement or parses an expression. All of the classes that are connected to the “Expression” class are the expressions that can be parsed, and the classes connected to the “Statement” class are different statements that can be parsed, with addition to the CatScriptProgram class, since this is where the program first starts.



We have also included a sequence diagram of parsing a PrintStatement that instead of giving a broad overview of the program, analyzes the workings of a single statement being parsed. The diagram looks like the following:



## Section 6: Design Trade-Offs

The major design trade-off that we pursued was using the recursive descent algorithm versus a parser generator. There are several reasons behind this decision. The main goal of writing the Catscript compiler was to function as an educational process. We wanted to implement a project

that would teach us more information about the topic. By using a recursive descent algorithm, we were able to directly implement the functional part of the parser, which required having a strong grasp of how the parser of a compiler functions. In contrast, using a parser generator directly conflicts with this goal, and does not require as much of an understanding of how a parser functions. While we could still have learned how a parser works, it would not have been achieved or even helped at all by using the parser generator.

Another major benefit of the recursive descent algorithm is ease of debugging. By writing the parser manually, we can track down errors and understand the code well enough to address them. A parser generator would produce code that is not as clear, meaning if there is an error tracking it down and fixing it would be far more difficult. Furthermore, we then are reliant on the parser generator's error messages, which if they are not clear results in an incredibly difficult process of debugging the parser.

## **Section 7: Software Development Lifecycle Model**

The model we used for this project was Test Driven Development (TDD). The benefits of using this approach were we had a clear vision of what needed to be implemented, and when it was implemented correctly. Once the tests concerning a particular aspect were passed, we knew it was safe to move on to the next area. For example, once all the tokenization tests were done, I moved onto implementing the parsing expressions part of the compiler. The tests also helped to highlight edge cases that we might have not considered while writing the code, leading to an overall more robust system.

On top of these benefits, TDD also offers benefits to the writer of the code as a human being. When programming the compiler, getting a test passing meant progress and that a level of success had been achieved. This makes the coding far more enjoyable, which equates to more motivation to work harder and longer. Without this, this progress would not have been as apparent, and it would have been more common to lose focus while coding and produce a less effective compiler.