

# Compilers Portfolio

---

Brendan Harrington

## Section 1: See Attached Source Code

---

<https://github.com/BrendanHar/csci-468-spring2023-private/tree/main/capstone>

## Section 2: Teamwork

---

### Testing

---

The Team that I was a part of for creating Catscript was composed of myself and Mitch Froelich as my partner.

While creating Catscript we dedicated time in the project to creating tests and analyzing each other's code to create documentation. This code review was an important part of our process, as it helped to ensure that the code is correct and sound. We focused on looking for syntactic errors, semantic errors, and optimization to improve each other's code.

Here are some of the tests that my partner wrote for my Catscript:

```
@Test
void capstoneTestOne(){
    assertEquals("2\n3\n", executeProgram("for( x in [1, 2, 3] ) {\n" +
        "var y = x\n" +
        "if(y > 1) {\n" +
        "  print(y)\n" +
        "}" +
        "}\n"));
}

@Test
void capstoneTestTwo() {
    assertEquals("6\nY is greater than 3\n", executeProgram("var x = 6\n"+
        "var y = 2\n"+
        "if(x<10){\n"+
        "print(x)\n"+
        "if(y>3){\n"+
        "print(y)\n"+
        "}" +
        "else{\n" +
        "print(\"Y is greater than 3\")\n" +
        "}" +
        "}"
    ));
}

@Test
void capstoneTestThree() {
    assertEquals(ErrorType.DUPLICATE_NAME, getParseError("var x : int = 10\n"+
        "var x = \"a\"\n"
    ));
}
```

### Documentation

---

The other way in which we relied on each other was through documenting each other's Catscript. This forced us to examine each other's code with a thorough analytic eye. This allowed us to catch more errors that may have been missed through our testing. This also meant that we would have to understand the overall structure behind the code that we created.

---

## Section 3: Design Patterns

---

```
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){
        return listType;
    } else {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return new ListType(type);
}
```

The Design Pattern that we chose to implement in Catscript was the memoization design pattern. The memoization design pattern is one in which values of computationally expensive operations are cached for future reference. We decided to implement the memoization design pattern for returning list types in Catscript.

This was done by first checking a hash table to see if there was already a key value pair for the list type of the specific list instance that the method is being called on. If no list type exists for this specific list instance, then a list type is placed into the hash map for future reference.

This is done to cut down on time spent looking up the list type of any instance of a list, and results in constant search time for getting the list type.

One of the drawbacks of using this design pattern is that the first time a new instance of a list type that is not in the table, will decrease the performance of the list type search. This is because for the first time a new list type is searched it goes through the extra steps of checking the hash table, creating the new list type, storing the list type in the hash table, and then finally returning the value. However, the upsides far outweigh the downsides as the performance hit only effects a new list type instance and every time after is more efficient than not memoizing list types.

## Section 4: Catscript Documentation

This document will demonstrate and explain basic syntax and declarations within the Catscript language.

### Introductory Information

Created in CSCI 468. Statically typed scripting language that is based around recursive decent parsing. Only six data types, and simple features with similarities to other programming languages, making it a great fit for basic language learning. Syntax similar to that of Java. Catscript is designed to be a simple and easy to learn scripting language.

Basic example of Catscript:

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

output: abc
```

### Data Types

Catscript has the following six data types:

- int - Catscript implements 32-bit integers.
- string - Stores standard text values. A string value is enclosed in double quotes in Catscript.
- boolean - This data type stores true or false values.
- list - Stores a collection of values of various types. A list in Catscript is immutable.
- null - This data type is used to represent the absence of a value.
- object - Similar to Java's Object data type. Can be used to store various types of data.

### Operators

Catscript supports the following operators:

- Arithmetic operators: (+, -, \*, /)
- Comparison operators: (==, !=, <, >, <=, >=)
- Logical operators: !

The only operator that is overloaded in Catscript is the + operator. Catscript's + operator is similar to Java's + operator as it also allows for string concatenation as shown below: `var x =`

```
"abc" var y = "def" print(x + y) output: abcdef
```

---

## Control Flow Statements

---

### If Statements

Catscript has if-else statements. The expression in the parenthesis is evaluated, and executed if the expression is in fact true. If the expression evaluates to be false, the if statement will be skipped and move onto an else statement where applicable. if-else statements contain an else statement. They are not required for an if statement to be followed by an else.

The syntax for an if statement is as follows:

```
if (<expression>){
  //Statements
}
```

The <expression> can be any valid Catscript expression that evaluates to a boolean value. For example, the following are all valid conditions:

- `x > y`
- `true`
- `name == "Hunter"`
- `year >= 23`

The else statement is option but will follow directly after an if statement as applicable.

The syntax for an if-else statement is as follows:

```
if (x > y){
  z = x + y
} else {
  print(x)
}
```

### Nested If Statements

You can also nest if statements inside other if statements to create more complex logic. The syntax for a nested if statement is as follows:

```
if (x > y){
  z = x + y
  if (z > 3){
    print(z)
  }
  else{
    print("False")
  }
} else {
  print(x)
}
```

### For loops

The for statement is another control flow statement used to iterate over values and execute a set of statements for each value.

The syntax for a for statement is as follows:

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

The <variable> takes on the value of each item in the <expression> during each iteration of the loop. The <expression> is any iterable data type, such as a list, that is collection of values to iterate over. Curly braces {} enclose the statements inside the for block.

---

## Other Features

---

### Variables

Variables must be created before their usage. The syntax for variable declaration is as follows:

```
var x = 4
var y = "abc"
```

Assigning Values to Variables:

Variables can be explicitly created by declaring the type after the variable name.

To create an explicitly declared variable you will use the following syntax:

```
<variable_name> : <type> = <expression>
```

### Variable Statement example

```
var day = 3
var month = "May"
var celebration = true
var year : int = "2023"
```

---

## Functions

A function in Catscript can be declared using the `function` keyword, followed by the name of the function, a list of parameters enclosed in parentheses, and an optional return type.

The syntax for declaring a function is as follows:

```
function <function_name>(<parameter_list>) [: <return type>] {
// Function body
return <return_value>;
}
```

The `<function_name>` is the name assigned to said function, and it should be noted it cannot share names with variables. The `<parameter_list>` is a comma-separated list that the function expects. Each parameter should have a name and can be explicitly declared. The types can be any of the six data types supported in Catscript.

### Parameter List:

The parameter list is a comma-separated declaration of expected values. If the parameter has an explicit type it should be declared by having the type declared after a colon following the name.

The syntax for a parameter with an explicit in a parameter list is as follows:

```
(<parameter_name> : <parameter_type>, ...)
```

### Return Statement:

A return a value can be used in a function to return a value. The return statement will usually be followed by the value that the function returns. If a function is to return a value, it must have a matching return value and expected return value.

The syntax for the return statement is as follows:

```
return <value>
```

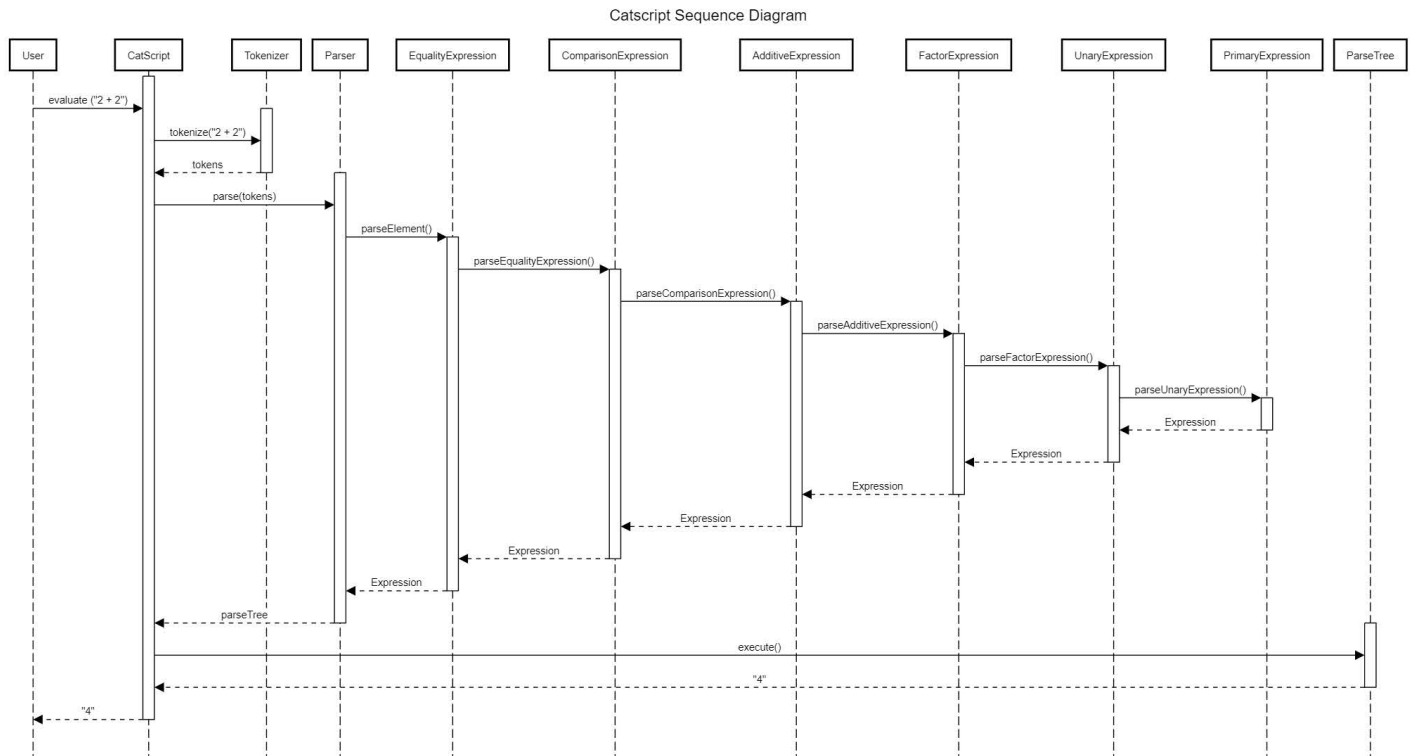
### Example Function

```
function foo(x: int, y){
  for (i in x){
    print(y)
  }
}

foo(23, "Hello, World!")
```

---

## Section 5: UML Diagram



---

## Section 6: Design Trade-Offs

---

### Immutable Lists

---

One design tradeoff that we faced when creating Catscript was having immutable lists. The reason that we made lists in Catscript immutable is to support covariance between the array type and the component types. One of the upsides that immutable have is that they provide enhanced safety and predictability compared to mutable lists, but at the expense of some flexibility and performance.

Immutable lists cannot be changed after declaration. This means that any function or operation that works with lists in Catscript can be certain that there will be no type errors when dealing with lists. Immutable lists are also easier for newer programmers to understand, since they have a fixed state that cannot be altered.

However, this safety and predictability comes at the expense of flexibility, ease of use, and performance. For example, if a method needs to change a list frequently, immutable lists are a horrible choice as the method would need to create a new list every time a single value is changed. This can result in increased memory usage and reduced performance.

Because Catscript is a small scripting language with little features, we decided that safety and predictability were a higher priority than performance, and in this case immutable lists were good choice to avoid some quirks of Covariant lists.

## Section 7: Software Development Life Cycle Model

---

### Introduction

In this project the software development life cycle model that we chose to use to develop Catscript was the Test-Driven Development Model. This model was chosen because there are many benefits that this model offered our project.

### Benefits

One of the reasons that we used Test-driven development (TDD) as our software development model was because TDD emphasizes writing automated tests before writing code. The main idea behind TDD is that if you write tests before writing code, you will end up with more reliable, maintainable, and bug-free code. When developing Catscript, using the TDD life cycle model helped in several ways:

#### Ensuring Requirements Are Met:

Writing tests before writing code helped to ensure that all the requirements of the language that were set at the beginning of the semester are met. By writing tests for each requirement and ensuring that the tests passed, we could be confident that the language was functioning to our requirements.

#### Catching Bugs Early:

By writing tests before writing code, any bugs or errors could be caught early in the development process. This allowed us to fix issues before they became more difficult to solve later on. This was especially important in creating Catscript as if errors from an earlier stage of the project like Tokenization were not caught, it would prevent parsing from working correctly even if it is coded correctly.

#### Improving Code Quality:

Writing tests before writing code also helped to improve the quality of the code. By ensuring that the code met the requirements and passed all the tests, we could be confident that the code was reliable and maintainable.

#### Facilitating Collaboration:

TDD facilitated collaboration in our team by allowing us to verify that each other's code worked beyond pre-existing tests. This meant that we had a chance to have the code tested from a perspective other than our own.

### Downsides

Since TDD involves writing tests before coding, we had to have a clear understanding of the structure of what we were creating before starting to write code. This also meant that changes to tests or code during the development process may require a significant rework of sections of code, or tests.