

# Program Instructions

---

Please include a zip file of the final repository in this directory.

## Section 1: Teamwork

---

### Collaboration

During the development of the *Catscript compiler*, the team focused on working together through certain portions of the project. The project had proper guidance and learning points to allow practice of tokenization, parsing, execution, and bytecode generation, which stopped some collaboration on the compiler. The collaboration done between **Jace (myself)** and **Ben H. (my partner)** focused on testing the compiler, working on documentation, and several other factors discussed below. The teamwork showcased below was equally distributed and allowed for multiple opportunities to troubleshoot and work through problems in a collaborative effort. Approximately twenty hours was used by both members, individually, to create the following test, documentation, and other items used in Catscript.

### Group Test Development

Each member of the group contributed test to challenge the Catscript compiler. The test were meant to focus on higher level elements of the language with internal *for loops*, complex *if statements*, and other advanced data structures for the simple systems built into the Catscript compiler and language. The team split up specific test and used responsive feedback to help both members work through the problems so no errors were thrown when using test driven development processes.

### Test Provided by Jace Z.

1. The following test showcases long *if statements* how many will statements will work in quick succession in the language. This test allows the user to see how many statement calls will challenge the Catscript compiler. The compiler has to execute many boolean checks and load individual scopes for each successful statement. The statement can be modified to showcase multiple outputs, singular outputs, or complete rejection of all *if statements*.

```
@Test
void longLongIfStatementWorks()
{
    assertEquals("3\n5\n", executeProgram(
        "var x = 3 \n" +
        "if(x == 1) { print(1) }\n" +
        "if (x == 2) { print(2) }\n" +
        "if (x == 3) { print(3) }\n" +
        "if (x == 4) { print(4) }\n" +
        "else { print(5)}"));
}
```

While testing the following test, my partner needed to ask no questions. However, the group collaborated to make sure that each *if statement* passed when operating the code. This means no feedback was necessary and the code worked to its fullest.

2. The following test showcases *if statements* within the *for statement* data structure, which will cause the compiler to yield multiple results over the lifetime and scope of the *for loop*. This test is a challenge to the compiler to make sure that variables are holding proper scope and are executing multiple, individual, instances of the same *if statement* structure. When passing the user will have verified the Catscript compiler can execute *if statements* within other statement and expression data structures from the Catscript language.

```
@Test
public void personalForIfLoop()
{
    assertEquals("false\nfalse\ntrue\n", executeProgram("for(x in [1,2,3])
{\n" +
        "if(x == 3){ print(\"true\") }\n" +
        "else { print(\"false\") } }"));
}
```

While testing, my partner needed some direction to make sure their scope was functioning properly during our collaboration. The test led us to his *for statement* systems that needed to have some scope modifications to the class itself. The scope was not terminating within the scoping validation section of the *for statement* class, which would prevent the code from allowing the variable to be accessible in other *if statements* and data structures.

3. The following test showcases *if statements* encapsulating *if statements* and a *for statement*, which helps to show more complex structures being nested within other Catscript data structures. We can use this test to make sure that an individual will run their compiler with proper scope and thread independency, but a user can easily mistake this test by not implementing proper formatting of the Catscript grammar. The test is built to run a *if statement* that will run other statements, which branches into further nested statements. If an individual can go all the way into executing this program then that will confirm the Catscript compiler is running the proper code structures needed to implement large and complex systems.

```
@Test
void personalIfForIfOutput()
{
    assertEquals("true\ntrue\ntrue\n4\ntrue\ntrue\ntrue\ntrue\ntrue\n",
executeProgram(
    "var hold = true \n" +
        "if(hold == true) { for(x in [1,2,3,4,5,6,7,8]) {\n" +
        "if(x == 4) { print(4) } if (hold == true) {
print(\"true\") } else { print(\"other\") } } }\n" +
        "else { print(\"failedHold\") }"));
}
```

**Test Provided by Ben H.**

1. The following test, designed by my co-developer, showcases how *if statement* data structures work within the *function statement* data structure. When troubleshooting the code with our compiler and language I had to ask for assistance with the return structure. My program caused errors that would halt the program for lack of a return output. The issue was resolved by tracing through the *function statement* to the *return statement* data class. The class need proper typing and scoping to help yield a proper output, which fixed the program by working through the proper test driven development style.

```
@Test
void customFunctionIfStatmentInFunctionWorks()
{
    assertEquals("1\n10\n", executeProgram(
        "function foo(y : int) : int {\n" +
            "if(y == 1){ print(2) }\n" +
            "else{ print(1)}\n" +
            "return y + 1\n" +
            "}\n" +
            "print(foo(9))"
    ));
}
```

2. The following test, designed by my co-developer, showcases how multiple parameter variables work within the *function statement* data structure. The test worked the first time on compiling and needed no collaboration to fix any present issues, the *return statement* fix helped to prevent any issues with the following test.

```
@Test
public void customFunctionWithMultipleVariablesPassedInForAndIfInside()
{
    String function = "function foo(b : int, c : bool) : int {if(c==true)
{c=true}else{print(\"works\")}}for(x in [b]){ print(x) } return b +2}\n";
    FunctionDefinitionStatement expr = parseStatement(function);
    assertEquals("works\n1\n3\n", executeProgram(function +
"print(foo(1,false))"));
    assertNotNull(expr);
    assertEquals("foo", expr.getName());
    assertEquals(2, expr.getParameterCount());
    assertEquals("b", expr.getParameterName(0));
    assertEquals("c", expr.getParameterName(1));
    assertEquals(CatscriptType.INT, expr.getParameterType(0));
    assertEquals(CatscriptType.BOOLEAN, expr.getParameterType(1));
}
```

3. The following test, designed by my co-developer, showcases a nested *if statement* within the *for statement* data structure. When running the test an error showed how the loop would only print once in our following test. With collaboration our group traced back through the *for statement* data class to

find that the scoping of the statement was not operational. This meant that the `x` value was initialized but did not key in our structure to print any besides the output `false`. We simply rearranged the scoping of our `for statement` variables so our output would yield everything needed, which caused the test to pass by implementing our test driven development techniques.

```
@Test
public void customForWithIfElse()
{
    assertEquals("false\ntrue\nfalse\n", executeProgram("for(x in [1,4,7]){
    if(x ==4){ print(\"true\") } else { print(\"false\") } }"));
}
```

## Assignment of Task

During the development process, I took the position of a lead developer that guided the design of key test in our development process. My co-developer took the position as a test developer in our collaboration, and our group total work time was forty hours of collaboration during the compilers project. The time was spent in the following manner: 20% Documentation, 60% Test Driven Development, and 20% Brainstorming Collaboration Periods. The development of individual test was shared and the documentation was spent separately through the duration of the class. The development of the compiler and language was unique and allowed thorough collaboration with our development model, but lacked in some error correction, troubleshooting, and design collaboration steps.

## Section 2: Design pattern

---

### Memoization Model

Memoization is a model that focuses on the design of efficient and less costly coding in a language. Some languages may experience burdensome typing, data manipulation, variable management, and other data structures that would cause the program to use much more memory and space. Memoizing a type, set of objects, variables, and more can be a form of caching in the language, which will help materials to be retrieved faster and without as much issue. The following code is not necessary for all languages but is used in Catscript to provide a more efficient typing system that can be used by the entire language.

### Catscript Type Memoization

In our Catscript compiler we implement the below Memoization for our Types in the language. The types would normally be individually set up and initialized before any types entered into the language. This means that we would write out specific slots of memory for initial type values, which is alright but could be cached as we approach types instead. This means that we will first create a static map that will hold the Catscript values defined by the type as its key and value.

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES
= new HashMap<>();
```

After we define this variable of maps, we will use that to be allocated throughout the running of our compiler. This means that Catscript will build its type list while running the compiler and will have only the needed memory to use in its type system. Below we can see that our types will take a list and will build new types when we see a null type entered, which will happen for the first instance of every type. This means that as we run the code and execute the possible test to launch the software, our system will build each new type and fill our Memoization map.

```
if (listType == null)
{
    listType = new ListType(type);
    LIST_TYPES.put(type, listType);
}

return listType;
```

This system is like the concept of caching data in a Redis or other form of cached area. The system is used in much bigger, data heavy, languages and provides proper memory usage throughout a programming compiler. We use this in Catscript to demonstrate a properly used design model, and to help make our type system more efficient for cumbersome functional programs that are created.

## Section 3: Technical Writing

---

### Catscript Guide

#### Introduction

Catscript is a fairly simple language, as it is only meant to teach the writer of the language how a compiler takes our input as a programming language and turns it into machine code that allows the computer to perform what we are requesting of it. Understanding compilers by creating one helps us better understand how programming works in general and the importance of choosing the proper programming language for the task you need to do.

Catscript is a strong, dynamically typed, functional programming language that was written in Java. While we have objects, they are limited on what they can do and are only used as types based on the Object class from Java, there is no inheritance, polymorphism, etc. Objects are only used to handle certain data types such as when we create a variable, we do not need to explicitly tell it what type of primitive data structure we are assigning it, although we can. In this documentation I will go over all the expressions and statements we can use in this language.

When using Catscript, there is no need to ensure we indent lines such as python or use a semi-colon after every command we want to run. Catscript automatically handles this and when we parse, it uses Extended Backus-Naur Form along with the tokenized keywords to evaluate your code and run it in order. This means that before you use a variable or function, it must be declared before you call it. This is because when it is run, it is put on the stack and runs from top to bottom in order. Example: `var x = (1+1-0) print(x)` Will evaluate successfully, but

```
print(x)
var x = (1+1)
```

Will not evaluate. This is important to remember when coding in a functional programming language such as Catscript. In Catscript, we created a compiler that works by recursive descent. After we use lexical analysis to generate our tokens, we can start parsing our program. What this does, to put it simply, is go down a big program, a class called CatScriptParser, and when a match is found for an expression or statement, it will either enter another iteration of the same program or return the evaluated expression or statement.

We can also handle errors related to parsing, determine the type, and then return that during runtime. Errors here are handled as expressions called Syntax Error Expressions. This can help us debug and find where the error is in our code. One of the ways we determine errors is by handling where statements and expressions start and end, that way, if there is an error in a particular place, it can tell us in what place the error is attached.

## Features

Comments: To comment out a line, use a double forward slash, or //. Everything on that line will not be evaluated when the program is executed.

## Catscript Types:

Catscript has seven different types. String, Integer, Boolean, List Literal, Object, Null, and Void.

- String type, or String Literal Expression: A string type must be typed out with quotations around it.

Example: "string"

To print out a new line, use \n in a string.

Example: var x = "line 1 \n line 2" print(x)

Will evaluate to line 1 line 2 If you need to put quotations inside of a quotation, then you must use \" around the text you are putting quotations around to define your quotations inside of the string.

Example: " \"string\" "

Say I wanted to create a new string variable with the use of quotations inside the string, then print x.

Example: var x = "\"string\"" print(x) Will print "string".

A String type is based on the String.class in Java.

- Integer type, or Integer Literal Expression: If the syntax is correct, any number typed in will be tokenized as an integer and is designated as "int". Integers can only use whole numbers and have a max value of 231-1, and a minimum value of -231. An Integer type is based on the Integer.class in Java.
- Boolean type, or Boolean Literal Expression: A Boolean type can either be true or false. When using a boolean it is specified as either "true" or "false".

```
For example: var x = true
              if (x == true) {print "true"}
```

```
else {print "false"}
```

When ran, the output would be "true". A Boolean type is based on the Boolean.class in Java.

- List type, or List Literal Expression: List is an type that can hold more than one integer, strings, Booleans, and objects at a time. When creating a list we only have to specify what the list will hold when creating a list of objects. A list of objects can also hold null and void types. Lastly, lists in Catscript are created as a linked list, this is important to note since it could cause issues if it runs out of memory on the stack.

For example, to create a list of integers implicitly: `var x = [1,2,3]`

For example, to create a list of integers explicitly: `var x list<int> = [1,2,3]`

For example, to create a list of objects: `var x = [1, "b", true, null]`

- Object type: An object type is based on the Object.class in Java.
- Null type: A null type is based on the Object.class in Java. Variables and objects in lists can be set to null.
- Void type: A void type is based on the Object.class in Java.

## Catscript Expressions:

Programming languages enable us to leverage the immense processing power of computers to accomplish tasks that would be impossible for humans to perform manually. This is the true potential of computers, allowing us to carry out millions of calculations in just seconds. This power has led to numerous technological advancements that have shaped the course of human history. At the heart of all computers lies their ability to perform mathematical calculations, and this is where expressions come into play. Once we have tokenized our code, we parse it out to understand the intended purpose of each word. All expressions in Catscript are extensions of the main expression class.

The types listed previously are created as expressions in Catscript except for object and void type. We have Boolean, Integer, List, Null, and String expressions. These are treated as expressions so we can validate them, store, and get their type, get values, evaluate, transpile, and compile. Each expression listed here doesn't all have the same functions.

Additive and other arithmetic expressions are used to evaluate simple calculations in Catscript. When testing, parentheses are not required to evaluate the expression, but when writing a program, it is required to use parenthesis, (), around them. Without them, parsing errors will occur. Lastly, they are read from right to left, so for example, when we divide, the dividend is on the left-hand side, and the divisor is on the right-hand side. Lastly, these expressions are primarily used to evaluate integers, but can be used on other types.

### Additive Expression:

An additive expression is used to take an integer and either add or subtract the values. We can perform as many calculations as you want when doing so.

Example: `1 + 1 - 1` Will evaluate to 1. Example: `var x = (1+1-0)` Will set the variable "x" to 2. With the parenthesis, it tells the parser where the additive expression starts and ends.

The next use of the Additive Expression is to concatenate string values to one another or to an integer value. It does not matter in what order you perform this operation, but you must use the plus, "+", symbol when concatenating.

For example: "What is 1 + 1?" + 2

Will evaluate to, What is 1 + 1? 2

### Comparison Expression:

A comparison expression is used to equate two different values and is mainly used in if-else statements. We can compare any two similar types. There are four different ways we can compare. When comparing, the right-hand side of the expression is considered the main value and the left-hand expression is what we are comparing it to.

```
Less than: <
Greater than: >
Less than or Equal to: <=
Greater than or Equal to: >=
Example: 5 > 4
```

This asks if 5 is greater than 4 and will evaluate to true.

Say for our next example we have: 5 <= 4

This will evaluate to false, since we are asking if 5 is less than or equal to 4, which is not true.

Like the Additive and other arithmetic expressions, we can evaluate simple calculations in Catscript without using parenthesis, but when writing a program, it is required to use parenthesis, ( ), around them. Without them, parsing errors will occur. For example: `var x = 5 if ( x >= 4) {print ("true")}` This will print out "true".

### Equality Expression:

An equality expression is used to check two values, with the same type, of equivalence. The options we have are either equal to or not equal to. This is similar to comparison expression as it is mainly used in if-else statements, return statements, and in lists with integers.

```
Equal to: ==
Not Equal to: !=
```

Example: `1 == 1`

Will evaluate to true.

Example: `1 != 1`

Will evaluate to false, since 1 is equal to 1.



**Factor Expression:**

A factor expression is used to multiply or divide integers.

To multiply, use an asterisk: \*

To divide, use a forward slash: /

Example: `print ( 5 * 5 )`

This will print out 25.

Example: `print (6 / 2)`

Will print out 3, as 6 is the dividend, or the number to be divided, and 2 is our divisor.

**Integer Literal Expression:**

An Integer Literal Expression is an expression that simply holds an integer value. With this, when we create a list, we can have a list of integer literal expressions. It is equivalent to a integer type in Catscript.

**List Literal Expressions:**

Otherwise known as lists, lists are a type of expression that holds a linked list of other expressions. This includes integers, strings, Booleans, and objects. An object can include any of the other mentioned expression types. To create a list, we need to use square brackets to enclose our list. If we are using a basic data type, such as integer or string, we do not need to designate the list as such when creating it.

Example of an integer list: `[1,2,3]`

To create a list of integers we can also designate the type, this only works with integers values, and it is not required.

```
Example: var list: list<int> = [1,2, 3]
for (x in list) {
  print(x)}
```

The output of the previous example will be, 1 2 3

To create a list of objects.

```
Example: var list = [(1+1), null, 3, true]
for (x in list) {
  print(x)}
```

This will create a variable, x, and create a linked list of objects that holds any type of expression. In slot 0, we have an additive expression, note the requirement for parenthesis around the additive expression, then a null expression, an integer expression and finally a boolean expression. The output of this previous example is 2 null 3 true. It is equivalent to a List type in Catscript.

**Unary Expression:**

This is used to either flip positive and negative integers, or to state the opposite condition of a boolean value. Again, since this is an expression, it can be used in a list of objects.

Example: `-1`

So, if we take the above example in a factor expression, then print the results. Example: `print(-1*2)`

The above example will evaluate to `-2`

To work with boolean values, we can use `not true`, to designate false. This is useful in if-else, return, and functions.

Example: `print(not true)`

The above example will return false.

**Catscript Statements:**

Catscript statements are the heart of most programming languages that allow us to start developing functioning programs and create our own custom data structures and algorithms with the use of these statements. Since Catscript is a simple functional programming language, we only have basic statements that get as complex as functions with multiple return statements and recursion.

**Assignment Statements:**

Assignment statements are only used to change the value of a variable after it has already been created as a variable statement.

```
Example: var x = "hi"  
x = "changed"  
print (x)
```

The output of this will be "changed". Since x was already created and designated on the scope, we can then assign a different value with the same type to x and change it.

**For Statements:**

For statements are used to iterate through a loop and through each iteration of that loop perform an action. For loops only iterate through lists, so if we use `[1,1,1]` as our list to iterate through, it will iterate through the for loop 3 times. The basic form of a for loop is `for (i in []) {inside for loop}`

Example: `for (x in [1,1,1]) { print(x) }`

The output of this for loop will be `1 1 1`

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

The output of this previous example is `1 2 3`. As we loaded in a list to iterate through that was previously created as variable x.

## If Statement:

The if statement is a conditional statement that simply says, "if something is true, do this", we can also have if-else statements, where if the if statement is not true, go to the else statement and execute. If statements have the basic structure of the keyword if, followed by parenthesis, (), where within the parenthesis our condition we are checking, and then if the condition is met, continue to what's inside of {}. Think of the conditions we are checking as an additive or similar expression, where we are comparing the values of two variables, integers, Booleans, etc.

Example: `if (5 > 4) { print (true) }`

The above example will print true, since 5 is greater than 4.

Example: `if (false == true){ print(1)} else {print (2)}`

The above example will print 2, since false is not equivalent to true.

Example: `var x = 9 if (x < 8) { print(true) } else{ print(false)}`

The output to the above example will print false, since x, which equals 9, is not less than 8, and the else statement will be executed.

## Print Statements:

The print statement is key in viewing what the output of our program was. It prints the value specified to the console window. The basic structure of a print statement is print(), where we can put any expression, or even a function call, inside of the parenthesis.

Example: `print(8)`

The output of the above example will print 8 to the console window.

Example: `print(1+1)`

The output of the above example will print 2, since 1+1 is an additive expression and evaluates to 2 when ran.

- Example: `function foo(): int { return 1} print(foo())` The output of the above example will print 1. We first have to create a function named foo, since this is a functional programming language and functions have to be created before we call them, and then when we call the print statement, it recognizes the function call based on the function name foo, plus empty parenthesis since we aren't passing any parameters into it, we just have to call it as foo(), and finally wrap it with the print statement, print(foo()).

## Return Statements:

Return statements are only used within functions. We can have multiple return statements in a function. These are usually separated with the use of if statements. We can return any type of expression.

```
Example: var x = 1
function foo(y:int): int {
    return y
}
print(foo(x))
```

The output to the above example will print 1.

```
Example: var x = 1
function foo(y:int): int {
    return y + 5}
print(foo(x))
```

The output to the above example will print 6.

### Variable Statements:

In Catscript, we have six different types that can be cast to a variable. These types are integer, string, boolean, list, null, and object. An object type can be of any of the basic variable types as well as null and void. An object is cast as part of the object class in java.

Using a variable statement, we have the option of explicitly or implicitly declaring the variable, except for an object variable, which must be explicitly defined.

- Example of a string variable: `var x = "foo"` The example above will create a string variable, without the need to specify `x` as a string. When the compiler tokenizes, it will determine that `"foo"` is a string, since it has quotations around it.

Below are a few examples of variables that can be cast without specifying their type.

- Example of an integer variable: `var x = 1`
- Example of a boolean variable: `var x = true`
- Example of a list variable: `var x = [1,2,3]`
- Example of a null variable: `var x = null` We can also specify what type we want the variable to be. This is required when we want to specify objects. After the naming of a variable, in this case `x`, we include a colon and then the type we want the type to be set as.
- Example of an object variable: `var x : object = "test"` This assigns the value `"test"` to `x` as an object and will determine that the expression is an instance of a string literal. When we assign a variable to an object its explicit type will be automatically determined.
- Example of an object variable with explicit type integer: `var x : object = 10` This creates a Catscript type object named `"x"` and then assigns its value as a java object that's explicit value is `"10"`.
- Example of an object variable with null as its explicit type: `var x : object = null` This creates a Catscript type object named `"x"` and then assigns its value as a java object that's explicit value is null.
- Example of an integer variable with type being defined: `var x : int = 1`
- Example of a string variable with type being defined: `var x : string = "value"`
- Example of a boolean variable with type being defined: `var x : bool = 1`
- Example of a list variable with list type also being defined: `var x : list = [1,2,3]` When assigning a variable to a list, refer to list literal expression under Catscript Types for more information.

```
For example: var x : list<int> = [1,2,3]
```

## Functions in Catscript:

Arguably the most important part of any programming language, functions allow us to create separate programs that can execute in countless ways and really bring our code to life. Catscript functions work basically the same as in most other popular programming languages, we just must format it properly. We can have an unlimited number of inputs and specify what kind of return value type it will be. The basic structure of a function is the keyword for a function, which is `function`, followed by the function name, parenthesis with the inputs we are sending into the function, and then the return type. All followed by the actual body of the function, wrapped in `{}`. To call a function, we need to call the function name, followed by parenthesis, `()`, and what inputs need to be sent into the function inside the parentheses.

```
Example: function foo() {return} foo()
```

In this example, we created a function called `foo`, passed no inputs, did not specify the return type, and in the body just returned nothing. To call the function, we just ran `foo()`.

```
Example: function foo(y : int) : int { return y + 1}  
print(foo(9))
```

In the above example, we created a function named `foo`, sent in a variable as an integer, specified what the function will refer to the variable sent in as, in this case `y`, and that it is an integer, the return value set to an integer. Then we have the body of the function, and a return statement. In this example the output will be 10.

```
Example: function foo(b : int, c : bool) : int{    if(c==true){c=true}  
else{print("works")}  
for(x in [b]){ print(x) }  
return b +2 }  
  
print(foo(1,false))
```

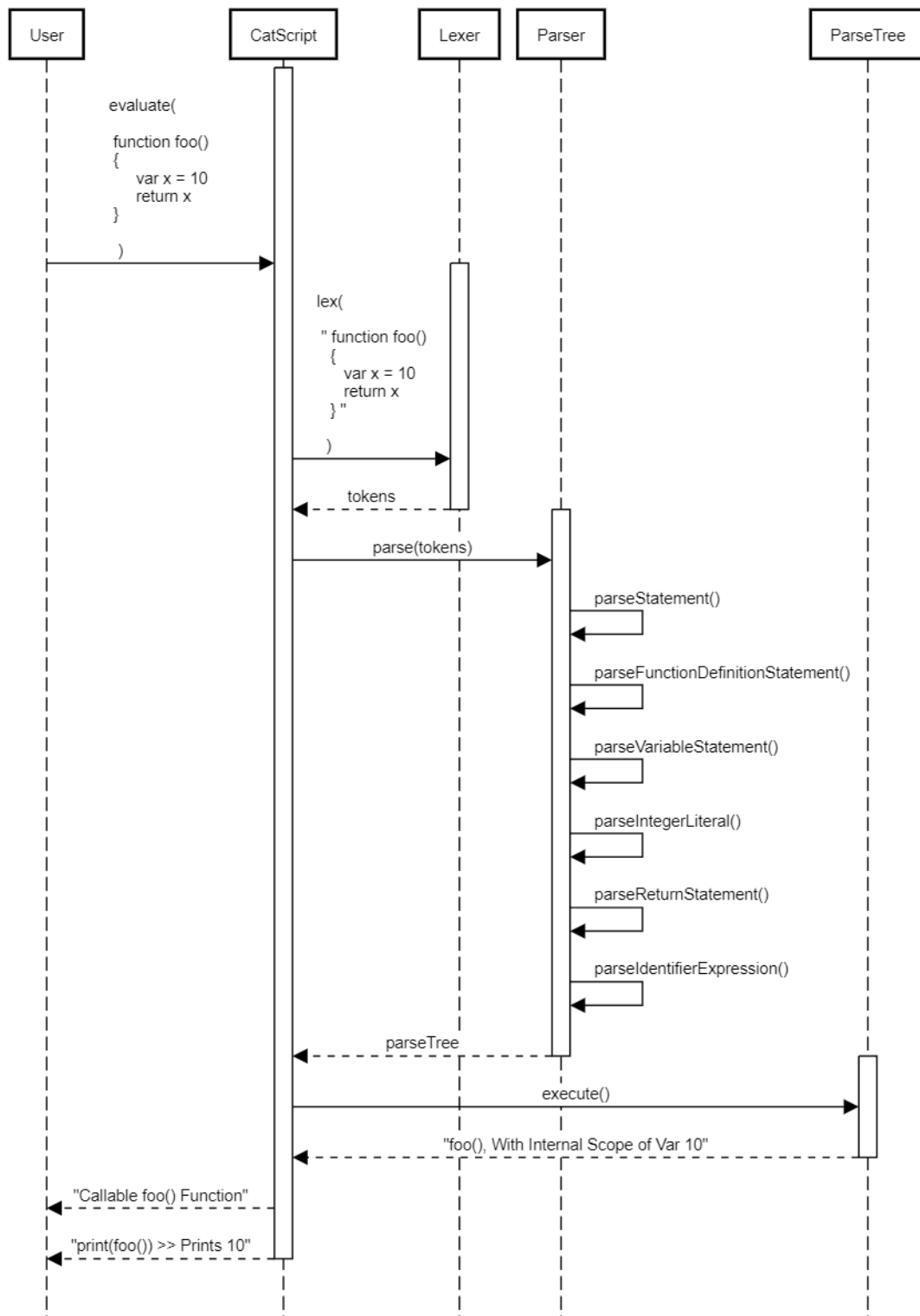
In the above example, the output will return `works 1 3` Notice how when we called the function, we sent in two variables, then specified in the function `foo` what we will call the variables inside the function, and that the return value is an `int`.

We can have a function return any of the basic types, and we can send in any expression into the function as a local variable. The local variable will stay inside of the function and will be destroyed once the function is done running.

## Section 4: UML.

The following sequence diagram is the UML showcasing how functions work within the Catscript compiler and language. A function is a large component to a functional language and allows many elements to be shown within the Catscript coding language. The UML was used to develop most of the language and focus on building functionality from normal processes completed within language, which is why no other UML will be used to demonstrate the Catscript language. The use of inheritance UML is too complex to demonstrate actual functionality and usability within the language, and the sequence diagram showcases much of the systems processes, statements, and basic input that would happen in Catscript.

Catscript Function Sequence Diagram



## Section 5: Design Trade-Offs

---

We created a parser by hand rather than using a parser generator. The parse generation tool would create a system quickly and would lack the models that are used in our system development. We learned direct tokenization, parsing mechanisms, and the execution of bytecode compilation, which all follows the systems of recursive descent.

Recursive Descent is the process of working from top to bottom when developing a language, which focuses on going from scanning input characters to running the bytecode of a new language itself. In the course, we learned how to build a proper context-free grammar that would frame out how our language would function, and the language started with scanning and tokenization. To tokenize our language, we followed the grammar to properly identify portions and label them accordingly as we tokenized each character, string, and value within our language. We then created a parsing system that would take the tokens and align their labels into individual scopes. The scoped labels and tokens would be aligned to a parse tree that would implement data structures like: *for statements*, *if statements*, *functions*, *variables*, and much more that exist in our language.

The final step of development took tokenized and parsed data, which is written in our languages format and syntax, and sent it to the JVM Bytecode systems to be properly executed. The code would compile with bytecode that we wrote and implemented into the language itself, and the overall design of recursive descent showed us how a language scans characters and implements them to a bytecode level compiling state. This design system shows much more than a parse generator would demonstrate, and after learning recursive descent I can look at any language and visualize how it scans through characters and strings to execute and compile at the lowest level of code.

Parse Generators are a mechanism, taught by most institutions, that allow you to input context-free grammar and output languages that compile down to another language's code or to a bytecode level. A Parse Generator will take a grammar file, a overview of the language itself, and a token definition that is filled with regular expressions, literals, and other elements to build the language. The generator will take the following information, based on an EBNF context-free grammar, and will compile it into a lexing program system. The lexer is a tool that will tokenize our inputs and prepare them to be sent to the parse generators created parser. A parser file will take another grammar file that specifies the name and general definition of the function of the language, which is going to be accompanied by several headers that define expressions, variables, statements, and any other information you wish to accompany the language. The two generated programs of a parser and lexer are then combined and connected to root of the hosting language, which is usually implemented by the parse generator. The programs combine and work with the languages bytecode to implement a language and use much of the visitor pattern design to link many classes together when generating the language. This method of language creation allows you to implement elements of a language.

quickly with complex regular expressions and EBNF that does not every get down to the lowest level of a languages implementation.

Both a Parse Generator and the Recursive Descent algorithm create a computer language that other developers can use in the computer science industry. A Parse Generator will take far less time to output a language and can be used to help bring isolation and uniqueness to the coding language that a company needs or uses. Although parse generators have much faster creation time, a recursive descent based language will benefit from the time taken to scan, scope, tokenize, and compile the language it creates. Recursive Descent takes time because you will scan all wanted characters, tokenize them to preferred values, scope and parse them into defined types by the grammars needs, and then use all of the gathered information to compile, or transpiling, into a given executed state. A recursive descent language will be able to transpiling a complex language for a company that wants to have more streamlined development in a fast language base, like C. Recursive descent will also show a developer how the input provided is processed and brought to the lowest level of execution, which can be useful for many avenues of coding development, unlike how a parse generator just gives you the language. The design trade-off between parse generation and recursive descent is very clear above, and a developer will more likely use the elements of recursive descent in their career than a basic parse generation that uses overly complex regular expression to make an unknown language.

## Section 6: Software Development Model

---

The development cycle of our programming language is Test Driven Development (TDD), which allows us to step through our language and solve problems, and bugs, as we approach them. This model creates faster development windows and allows us to quickly debug the system after an error is introduced. The error has clear locations and possible solutions when developing the language under TDD, while this model is good for problem solving, it makes the language a bit more complex to create initially. More rules are in place to make sure that less errors appear and the model does help to build a specific computer language for our project. The complexity is taught via recursive decent, and allows the learner to understand how a computer language works from bytecode to coding. This means that our TDD system will show the full flow and creation of a compiler, rather than creating languages via generators and automated systems. The rules are general principles in a compiler, which are enforced by the class curriculum and teaching methods, but a learner will be able to implement more complex architecture due to the environment that Test Driven Development creates.

TDD also helps the single and group development process, but does limit the flexibility to create new functions within the language without building new test for those systems. This model helped our team development on the project, which is due to the need for members to build new test while adding new functions, expressions, and methods to the language itself. Each member of a team would create a new function, while other members would use test to verify the new functions are working before launching the language to our compiler. This also works the same for new and wanted functions in the language, which are created via the guidance of a test built by individuals in the team environment. The same method can be done by one individual to create new expressions and statements in the Catscript language. TDD model allows the development of Catscript to be streamlined and builds collaboration with developing new test for the development process.