

## Section 2: Teamwork

For the teamwork section. I created three new tests for my partner to verify that their Catscript compiler works as it should. This is called quality assurance. I will first go through the tests I created and ran and then the tests my partner created for me to test my compiler as well. I try to go through what is happening during each step and what it can tell us is happening, such as a function statement or a list expression being created and running successfully.

The first test I created was to test an if-else statement in a function. This program takes a print statement, sends a function call with the number nine, and then once in the function, if the variable that was passed in is equal to 1, print 2, otherwise, print 1. Once the if-else statement has been executed in the function, it returns the value we passed into the function and prints the result of that function we originally passed into the print statement. This test was implemented in "CatscriptFunctionArgsAndReturnEvalTest", and the code for the test is as follows.

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

The second test that I created for my teammate was to test a function with multiple input variables ran through an if-else and a for statement. This takes a function as a string, and then parses the function, as a function definition statement, and then executes that statement. The variables we put in are an integer, known as variable b, and a Boolean, known as variable c. The return type of the function is an integer. Once the variables are passed into the function it first goes through the if-else statement, testing whether variable c is true or false, and then prints accordingly, then goes to the for loop, in which it only runs through one iteration and prints the value of our integer b. The last part is we return our integer b with two added to it. This test was implemented in "CatscriptFunctionArgsAndReturnEvalTest", and the code for the test is as follows.

```

@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));
}

```

These previous two tests show us that we are able to successfully tokenize, parse, and then execute a function, if-else, for, and print statement; with our function statement being assigned the name foo. This also determines that a if-else statement can properly handle conditions, that are implemented as a list of statements, and a body of set expressions. The last piece shows a successful parse return statement that can return an additive expression, and then finally print out the value returned from the function.

For our third test, I tested to see if a for statement could handle a list expression of integers, that would iterate through the number of elements in the list, and then use an if-else statement to determine if it was iterating through it in the proper order as well as proper amount of iterations. We send in a list literal expression with three integers in it and are able to successfully print three times based on the conditions of the if-else statement.

```

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

```

My partner provided three tests for me to run. I was able to pass all three of his tests. Down below are the tests. I will also describe how they ensure parts of my compiler are working successfully as I did with the tests I created.

The first test my partner created was a test that would create a variable and assign it a value of three. Then we take the variable, `x`, and run it through a series of if statements, with the last one being an if-else statement. When the variable is created and assigns it the value of 3, it successfully determines that 3 is an integer. Next, we go through the if statements and if `x` is equal to the specified variable to test against, it will print out that variable. Since we have multiple if statements, it successfully checks on the third one that `x == 3` and then prints 3. This tells us that our tokenizer was successful in scanning, then we were able to create a variable statement as well as a series of if statements, and it followed it in successive order. Since we have multiple if statements, where if a condition isn't met it doesn't go into that if statement to perform its designated functions, it continues on and keeps running. We know that it does this successfully since our last if statement is an if-else statement that successfully determines that since our variable `x` is not `== 4`, it has the else condition where it prints out the value 5.

```
@Test
void longLongIfStatementWorks() {
    Assertions.assertEquals("3\n5\n",
this.executeProgram("var x = 3 \nif(x == 1) { print(1) }\nif (x
== 2) { print(2) }\nif (x == 3) { print(3) }\nif (x == 4) {
print(4) }\nelse { print(5)}"));
}
```

The second test to look at takes a for loop and runs through the loop the proper number of times and then using if-else statements determines if the number being tested, in this case `x`, which is in a list, is equal to a value, it prints "true", if the condition is not met, it prints "false". This test determines that we can successfully tokenize the syntax, then create a for statement, take a list, and iterate through the list, assigning the value of `x` to each variable in the list as we iterate through the list. Since in the for statement, we are testing to see whether the value of `x` is equal to 3, it iterates through 3 times due to there being three digits in the list, and then determines that the first two values are not equal to 3 in our if-else statement, and prints "false" twice, then on the last iteration, since the value of `x` is assigned to the integer 3, the if statement properly determines that the condition for

the if statement has been met and prints "true". This shows us that we can successfully create list expressions, for statements, and if-else statements.

```
@Test
public void personalForIfLoop() {
    Assertions.assertEquals("false\nfalse\ntrue\n",
this.executeProgram("for(x in [1,2,3]) {\nif(x == 3){
print(\"true\") }\nelse { print(\"false\") } }"));
}
```

The final test created by my partner assigns the Boolean value true to "hold", and then checks with a if statement if the variable "hold" is equal to true, then go into the if statement and iterate through a for loop. This test shows us that a variables type can be successfully determined without having to define its type, meaning that it is a dynamically typed programming language when we check to see, using an if statement, if the value "hold" is equal to true. Once inside the if statement, it has a for loop to iterate through, since there are 8 values in the for loop, it iterates through 8 times, in the for loop, it has an if statement where it tests if the value "x" is equal to 4, if not then it goes onto the next part which is an if-else statement, where if the value of "hold" is equal to true, then print true. In the for loop, we are iterating through a list expression and assigning it to x based on its position in the for statement. This successfully shows us that, again, a list expression is being created properly and we assign the variable, in this case "x" to the number that it is at in its iteration of the for loop. Since we are iterating through the loop 8 times, when x is equal to 4, it prints 4 and then goes onto the next if-else statement, and since the conditions of the if-else statement are always true since it is checking the value of our variable "hold", it will always print true. Like the previous tests, it tells us that we have properly implemented an if and if-else statement, a for statement, list expressions, and variable statements as to the specifications of the Catscript language.

```
@Test
void personalIfForIfOutput() {

Assertions.assertEquals("true\ntrue\ntrue\n4\ntrue\ntrue\ntrue\ntrue\ntrue\n", this.executeProgram("var hold = true \nif(hold ==
true) { for(x in [1,2,3,4,5,6,7,8]) {\nif(x == 4) { print(4) }
if (hold == true) { print(\"true\") } else { print(\"other\") }
} }\nelse { print(\"failedHold\") }"));
}
```

Our team worked on the capstone project by mainly creating three tests for each other to try and also working on proper documentation.

Describe how your team worked on this capstone project. List each team member's primary contributions and estimate the percentage of time that was spent by each team member on the project. Identify team members generically as team member 1, team member 2, etc.

---

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

# 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:**

As long as 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  $2^{31}-1$ , and a minimum value of  $-2^{31}$

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.

Similar to 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 an 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 or 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 a 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** thats 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** thats explicit value is null.

Example of an **integer** variable with type being defined: `var x : int = 1`

Example of an **string** variable with type being defined: `var x : string = "value"`

Example of an **boolean** variable with type being defined: `var x : bool = 1`

Example of an **list** variable with **list** type also being defined:  
`var x : list<int> = [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 have to 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.