Section 4: Documentation:

Expressions and Operators:

Additive Operators:

Plus operator

```
1+"a"
```

The additive expression takes a right and left hand expression as well as an additive operator. The additive expression is left hand associative, so it will execute the two left most expressions if there are more than two expressions and no parenthesized expressions. The result will be the addition of the left and right hand expressions.

Minus Operator

```
1-1
```

The additive expression takes a right and left hand expression as well as a minus operator. The additive expression is left hand associative, so it will execute the two left most expressions if there are more than two expressions and no parenthesized expressions. The result will be the subtraction of the left and right hand expressions.

Unary Operators:

Minus operator

```
-
```

The minus operator in unary expressions acts as a negation of the right hand expression.

Not operator

```
not
```

The not operator in unary expressions acts as a bang operator on the right hand expression, which in turn flips the boolean value of the right hand expression.

Factor Operators:

Star Operator

```
1 * 1
```

The star operator is used to multiply a left and right hand expression, it returns the value as a single expression.

Slash Operator

```
1 * 1
```

The slash operator is used to divide a left and right hand expression, it returns the value as a single expression.

Equality Operators:

Equals Operator

```
1 == 1
```

The equality operator takes a left and right hand expression and compares whether they are equal. If they are equal, the expression will return true else it will return false.

Bang Equals Operator

```
1 != 1
```

The bang equality operator takes a left and right hand expression and compares whether they are not equal. If they are not equal, the expression will return true else it will return false.

Comparison operators:

Less than Operator

```
1 < 1
```

The less than operator compares two expressions and checks if the left hand expression is less than the right hand expression, if this case is true the expression returns true, else it will return false.

Greater than Operator

```
1 > 1
```

The greater than operator compares two expressions and checks if the left hand expression is greater than the right hand expression, if this case is true it returns true, else it will return false.

Greater than or Equal to Operator

```
1 >= 1
```

The greater than or equal to operator compares two expressions and checks if the left hand expression is greater than or equal to the right hand expression, if this case is true the expression returns true, else it will return false.

Less than or Equal to Operator

```
1 <= 1
```

The less than or equal to operator compares two expressions and checks if the left hand expression is less than or equal to the right hand expression, if this case is true the expression returns true, else it will return false.

Type Literals:

```
types: 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression, '>']
```

Integers

```
1
```

Integers literals in CatScript are integer values, with type INT (32 bit).

Strings

```
"string"
```

String literals are string values within quotations.

Boolean

```
true
```

Boolean literals represent true and false values that may be used to validate expressions.

List

```
[1,2,3]
```

List literals take a number of expressions that are seperated by commas and placed in between brackets. The list may be of many types, but they must be objects they cannot be declared as separate types otherwise an incompatible type error will bwe thrown.

Function Call Expression:

```
foo(1,2,3)
foo()
```

Function call expressions make a call to an expression with any number of arguments. Calling functions starts with using the correct identifier for the function you wish to call as well as closing the parameter list with left and right parenthesis. Declaring explicit types is not necessary when calling a function as the function declaration has already declared types, as long as the correct types are entered in their respective locations in the parameter list an INCOMPATIBLE_TYPE error shouldn't arise. If an UNTERMINATED_ARG_LIST error is shown this would imply that there is a missing parenthesis at the beginning or the end of the parameter list.

Literal Expressions:

Identifier Expression:

```
x, y, foo, etc.;
```

Identifier expressions are used to validate and or set the names of identifiers in assignment statements, functions, and function calls. When an identifier has been used more than once in the same scope it will return a DUPLICATE_NAME error.

Int Literal:

```
1, 2, 3,..., 9;
```

Int literal expressions are represented by any integer in the range 1-9. Int literals may be used as parameters, arguments, and expressions. These expressions also aid in implicit typing, the type int will be assigned to an identifier if there is no explicit type defined and the assigned value to the identifier is and integer in the range 1-9.

Null Literal Expression:

```
null
```

The keyword "null" is used to express an absence of meaningful value or to check for the presence of a value. Null may be used in conditional statements to verify that there is an expression representing an identifier before entering parsing a branch of an if statement. Null may also be used with logical operators to again check the validity of an expression. When using null there is a chance that a NullPointerException will arise if a type is defined as null and is called somewhere else in the program where the method is expecting a return value that is not void.

List Literal Expression:

```
[1,2,3]
[[1,2],3]
```

List literals may contain any number of elements as long as they are of valid types. List literal expressions may include int, bool, object, and null types, although you cannot declare explicit types within a specific index, therefore lists will be implicitly typed. Lists may be used in var statements, assignment statements, and for loop expressions. Errors may arise if a list is left unterminated.

Parenthesized expressions:

```
1+(3-1)
```

Parenthesized expressions are used to allow precedence control over certain operations.

Boolean Literal Expression:

```
true
false
```

Boolean literal expressions are comprised of the "true" and "false" keywords. They are also used to check validity in boolean statements. They are also used in expressions in if statements to check which branch of an if statement should be parsed. Logical operators may be used with "true" and "false" keywords.

Statements:

Print Statement:

```
print(1)
```

Print statements use a "print" keyword as well as an opening and closing parenthesis to encapsulate the expression. Valid types of a print statement are int, bool, null, and object. If the syntax of the print statement is correct it will return the print statements expression which in this case "print(1)" would be 1. If there is a syntax error such as an unterminated argument list it will return an error.

Var Statements:

```
var x = 10
var x : int = 10
```

Variable declaration statements are used to declare variables within a scope whether it be local or global. Variables may be declared both implicitly and Explicitly. Variable Statements require a "var" keyword as well as an expression on the right hand side. To explicitly declare a variable simply use the format:var x : int = 10, for object you would use the object identifier, for booleans the bool identifier and so on. Declared values may be used within it's scope or in a scope lower than it's current scope.

For Statements:

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

For statements require a "for" keyword, an expression that will hold the arguments and a statement where the executing of the method will occur. The following requirements are required in the argument as follows, ("identifier" in "expression"), identifier will be the name of your counter variable and the expression is representative of the number of iterations the loop will run for. Errors may arise if there is an unterminated argument list.

Assignment Statements:

```
x=10
x=null
x=true
```

Assignment statements require the identifier of an already declared variable as well as a right hand expression. The assignment statement will re define the value of a specific variable. These are type sensitive statements so yuo must set the new value equal to something that matched the type of the previous value.

Function Definition Statements:

```
function foo() : bool {
var x = true
return x
 }
```

Function definition statements require a "function" keyword, a list of parameter types, an optional return type, and a body where the return statement will be if there is a return value. The return value if used must match the return type declared in the function definition statement. The name of the function as well as the definition will be stored in a hashmap where it can later be referenced to ensure that there are no duplicate function and or variable names being used later in the program. If a duplicate name is used a duplicate name error will occur.

Function Call Statement:

```
foo(1,2,null)
```

Function call statements require an identifier that is referring to an already defined function, as well as a list of required parameters as defined in the function definition. Function calls will execute the body of a function using the inputted and give a return value if a return value is necessary. If a function call is used and a starting or closing parenthesis is missing it will cause an unterminated arg list error to occur.

If Statements:

```
if(x > 10){
    print(x)
 } else {
    print( 10 )
 }
```

If statements require an "if" keyword followed by an expression followed by a statement, and if necessary it may be followed by an else branch. The expression used to determine the branch that is executed should be an equality, comparison, or unary expression, otherwise there will not be a boolean literal return value and the if statement will never execute. The same parsing errors may occur here as function calls, if the expression is unterminated an unterminated arg list error will occur and if there are unterminated body statements an error will occur there as well.

Return Statements:

```
function x() : int {
    return 10
 }
```

Return statements require the "return" keyword as well as a return value that agrees with the type of the function. In the example the function is being defined with a return type of int and is returning type so that is a valid return statement. A return statement may return any type as long as the return statement and the type of the definition match.