

Capstone Portfolio

CSCI:468

Project Name:

Catscript

Team Members:

Christopher Danielson

Bandon Marceau

Instructor:

Carson Gross (Sponsored by JetBrains)

Section 1 - Code:

The code will be attached in a zip file

Section 2 - Teamwork:

The way that I worked with my team member is Brandon made tests for the project and made documentation for Catscript. I contributed all of the code for the tokenizer, parser, evaluations, and compilation of Catscript. I also contributed tests and documentation for Brandon Catscript project. The amount of time that I spent developing the code for Catscript was around 200 hours this semester. The amount of time that I spent making tests and documentation for Brandon's code was around 10 hours. The amount of time that Brandon spent making tests and documentation for my code base around 10 hours.

Section 3 – Design Pattern:

A design pattern that was used in the project was the memorize pattern. This pattern is used in the CatscriptType class in the parser directory in the project. The reason that I used this pattern was to get rid of a bunch of function calls to get a type and instead cache the types so they could be looked up faster. I put a comment above the design pattern pointing it out and included the code below.

Code:

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = cache.get(type);
    if (listType == null) {
        listType = new ListType(type);
        cache.put(type, listType);
    }
    return listType;
}
```

Section 4 – Documentation (Technical Writing):

Introduction:

Catscript is a simple statically typed language, meaning that when a type is declared for a variable it remains that type through the execution of the program. Boolean expressions, 32-bit integers, java style strings, the null type, and objects are all featured types in Catscript. Catscript also has lists that can be of mixed type and contain any of the types, even allowing for lists containing lists. Doubles and Floats are types that are not included in Catscript. Catscript has a variety of statements and expressions available in the language that will be discussed in further detail below.

Catscript allows for mathematical operations such as addition, subtraction, division, and multiplication as well as the ability to parenthesis mathematical expression to force order of operations. The plus operator can also be overloaded to allow for string concatenation. Catscript also has equality checking using two equals signs, as well as checking for non-equality with the bang-equal (!=) operator. Catscript does not include a modulus operator.

Catscript allows for commenting out sections of code by using two forward slashes (//) followed by whatever comments need to be written. Commented out sections of code will be ignored by the compiler. The following is an example of a comment.

```
// Here is a comment, it will ignored by the compiler
```

Expressions:

Expressions in Catscript evaluate to some value. They may be evaluated to a literal value or an object value. Integers, Strings, Booleans, the Null type, and Lists are all types that can be evaluated to a literal. Each of these literal types are documented below.

Integer Literal Expressions:

Integer literals are a 32 bit value. Integers can be easily implemented in Catscript by just typing in an integer. The following examples is a program that will evaluate to an integer literal:

```
1
```

This will evaluate to 1.

Note that a decimal number as follows will NOT evaluate in Catscript.

```
1.2
```

DOES NOT EVALUATE

String Literal Expressions:

String literal expressions are a series or array of characters. A string literal expression could be a single character, a word, a sentence, or several sentences. To implement a string in

Catscript it must begin with double quotations and then be closed with double quotations (single quotations will NOT work). Following are examples of how to implement a string.

```
"A"
```

Will evaluate to A

```
"This is a string"
```

Will evaluate to This is a string.

Boolean Literal Expressions:

Boolean literal expressions are evaluated to either true or false. Following is an example of Boolean literal expression.

```
true
```

Will evaluate to true.

Null Literal Expressions:

Null literal expressions have no value to them and can be declared with the null keyword. The following is an example of a null literal expression.

```
null
```

Will evaluate to null.

List Literal Expressions:

List literal expressions are similar to lists in the Python programming language, or arrays in other programming languages. Lists are made using a left square bracket to start the list and a right square bracket to terminate the list. Elements within lists are separated by a comma. Values could be integers, strings, Booleans, the null type, or even a list inside the list. The following is an example of a list with all values of the same type, integers in this case.

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

Will evaluate to [1, 2, 3]

A list could also include mixed data types as in the following example that contains an integer, a string, a null type, and a Boolean all within the same list. Catscript will assign list typing for mixed lists based on the highest order of elements within the list. The following will be seen by Catscript as a list of objects.

```
[1, "string", null, true]
```

Will evaluate to [1, string, null, true]

A list can also have lists within it as the following example shows.

```
[[1, 2, 3], ["this", "string"]]
```

Will evaluate to [[1, 2, 3], [this, string]]

Unary Expression:

Unary expressions are expressions that only contain a single operand, “-“, in front of an integer. This allows for the creation of negative values integers. It also allows for a user to negate truth values by using the “not” keyword. The following examples shows how the unary expression is used.

```
-1
```

Will evaluate to -1.

```
not true
```

Will evaluate to false.

Additive Expression:

Additive expressions include both addition and subtraction. Addition can be done on integer values, or it can be overloaded with strings to do string concatenation. Subtraction can only be done if both the left hand and right hand side of the expression are integers. The following example shows how integer addition evaluates.

```
1 + 1
```

Will evaluate to 2.

Addition and subtraction can be used as many times as necessary to create a longer expression that will evaluate from left to right as the following examples shows.

```
3 + 3 - 6 + 3 - 3
```

Will evaluate left to right consuming tokens until the final evaluation that will be 0.

The following shows a couple examples of string concatenation when the operator is overloaded. One shows an integer concatenated to a string, and the other shows two strings concatenated together.

```
1 + "string"
```

Will evaluate to 1string

```
"this" + "string"
```

Will evaluate to thisstring

The following example shows the subtraction operator, which only works on integers.

```
1 - 1
```

Will evaluate to 0.

Factor Expression:

Factor expressions include the multiplication and division operations on integers. These operations only work if the left-hand side and the right-hand side are both integers. The following examples show how multiplication and division can be called and how the expressions evaluate.

```
1 * 5
```

Will evaluate to 5.

```
10 / 5
```

Will evaluate to 2.

Multiplication and division can be used as many times as necessary to create a longer expression that will evaluate from left to right as the following examples shows.

```
3 * 3 / 9 * 3 / 3
```

Will evaluate left to right consuming tokens until final evaluation which will be 1.

Parenthesized Expressions:

Parenthesized expressions are expression with parentheses around them. Multiple sets of parentheses can be used in an expression. This can allow for a user of Catscript to separate out mathematical expressions to determine order of operations. There must be an equal number of opening and closing parentheses in the expression for it to evaluate. The following examples show uses of the parenthesized expression.

```
(( (1) ))
```

Will evaluate to 1.

```
(1 + 2) * 3
```

Will evaluate to 9. The parenthesized part (1 + 2) will evaluate to 3, then it is multiplied by 3.

Equality Expression:

Equality expressions are used to determine whether two expressions are equivalent by using the equal equal operator (==), or they can be used to determine if two expressions are NOT equivalent using the bang equal operator (!=). Equality expressions will return either true or false. The following examples show how this comparison can be done.

```
1 == 1
```

Will evaluate to true since the two integers are equal.

```
1 != 1
```

Will evaluate to false since this is asking if 1 is NOT equal to 1.

Boolean values and null types can also be compared.

```
true == null
```

Will evaluate to false.

```
true != null
```

Whereas this would evaluate to true since the Boolean true is not equal to null.

Comparison Expressions:

Comparison expressions check for a greater than, less than, greater or equal to, less than or equal to condition using the >, <, >=, and <= operators. This expression can only be done on the integer data type. The comparison expression will evaluate to a true or false based on the expressions being compared. The following examples show the comparison expressions in action.


```
5 > 1
```

Will evaluate to true since 5 is greater than 1.

```
1 > 5
```

Will evaluate to false since 1 is not greater than 5.

```
1 < 5
```

Will evaluate to true since 1 is less than 5.

```
1 >= 1
```

Will evaluate to true. This is checking if 1 is greater than OR equal to 1.

```
1 <= 1
```

Will also evaluate to true. This is checking if 1 is less than OR equal to 1.

Function Call Expressions:

Function call expressions can be used in Catscript to call a pre-defined function. Function call expressions can return a value that can be of any data type to a variable. There may be no return value. Parameters need to be passed into a function if that function requires. Parameters passed into a function using a function call expression will only be available in the scope of the function that is being called. The following examples show some function calls.

```
x = foo(1)
```

The variable x will be set to the return value of the foo function

```
foo()
```

The foo function is called with no parameters.

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

The foo function is called, in this case there were parameters sent in.

Syntax Error Expressions:

Catscript can identify errors that occur in expressions and add the error type to the parse tree. The following examples shows an unterminated list (missing a closing square bracket).

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

An error type `ErrorType.UNTERMINATED_LIST` would be added to the expression.

Statements:

Print Statements:

Print statements in Catscript use the print keyword followed by parenthesis. The data that is input inside the parenthesis will be displayed on the console. The following example shows how a string could be displayed to the console using the print statements.

```
print("Print this string")
```

Print this string will be displayed to the console as output.

A variable that has been assigned to a value can also be output to the console as shown in the following example.

```
var x = 10
```

```
print(x)
```

10 will be displayed to the console as output.

Variable Statements:

Variable statements in Catscript allow the assignment of a value to a variable that can be named by the user. The value of variables can change throughout execution of a program. Variable can be created with the 'var' keyword followed by a variable name of the user's choice. There are two ways that the data type of the variable will be assigned during variable declaration as can be seen in the following examples.

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

In this example the variable x is explicitly set to the integer type.

```
var x = 10
```

In this example the variable x is not explicitly stated as integer type, but it will be assigned the integer type by referencing the right-hand side as determining that it is an integer.

Assignment Statements:

Assignment statements can be used to assign a different value to a variable that has already been created. This will change the value that is stored in the variable. The example below shows the assignment statement in action reassigning the value of a created variable.

```
var x = 10
```

```
print(x)
```

```
x = 20
```

```
print(x)
```

The first instance of print(x) will show 10 on the console. When x = 20 is called, the value stored in x will change to 20. The second instance of print(x) will then show 20 on the console.

Note that although variables can change, the data typing of the variable cannot be changed using the assignment statement. The following example would NOT work in Catscript (Catscript is NOT dynamically typed).

```
var x = 10
```

```
print(x)
```

```
x = "string"

print(x)
```

This will cause a parse error, because Catscript is statically typed!

For Statements:

For statements iterate through a list of values. For statements can be declared using the 'for' keyword followed by a list of values inside parathesis. After the closing parathesis an opening curly brace is required to begin the body of the for statement. Following are a few examples of for statements in Catscript.

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

    print(x) }
```

The print statement will output 1 2 3 to the console.

```
for (x in ["foo", "bar", "for"]) {

    print(x) }
```

The print statement will output foo bar for to the console.

Nested for loops are also allowed in Catscript as the following example shows.

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

    for (y in [1, 2]) {

        print(x + y) } }
```

The print statement will output 2 3 3 4 4 5 to the console.

If Statements:

If statements in Catscript allow for a given Boolean condition to be checked before running a block of code contained within. If the condition is true, the code block within the if statement body will be run, whereas if the condition is false, the code would not be run. If statements can also have an else statement that accompanies them allowing for the code in the else statement body to be run if the if conditional evaluates false. The else block will be ignored if the if portion returns true. The following examples show how if / else statements work.

```
Var x = 10
if (x > 5) {
    print("x is greater than 5") }
else {
    print("x is less than 5") }
```

Will print to the console x is greater than 5, since the value of x is set as 10. The else portion is ignored.

```
var x = 4
if (x > 5) {
    print("x is greater than 5") }
```

```
else {  
    print("x is less than 5") }
```

Will print to the console x is less than 5 since the value of x is 4. The else portion runs since if portion is false.

Function Call Statements:

Functions in Catscript can be created using the 'function' keyword followed by the name of the function followed by opening and closing parentheses. Inside the parentheses are the parameters that need to be taken in by the function. A function can have no parameters, one parameter, or several parameters. When a function takes in parameters, those parameters then only exist in the scope of that function. Function can also have a return value as well but having a function return something is not necessary. Return statements will be described in the next section. Following is an example of a function that does not have a return statement.

```
function isGreaterThanFive(x : int) {  
    if(x > 5) {  
        print("Yes")  
    }  
    else {  
        print("No")  
    }  
}  
  
isGreaterThanFive(10)
```

The bottom line `isGreaterThanFive(10)` sends the value 10 into the function as the parameter x. Since x is given the value 10, the function would print Yes in this case.

Return Statements:

Return statements are statements that must reside at the end of a function. Return statements return a value from the function that can be used elsewhere in the program. The return statement allows for any Catscript type to be returned from a function. The return type should be specified in function creation with a colon and return type after the list of parameters. The following example shows how a function with a return statement works.

```
function returnString() : string {  
    return "a string returned"  
}  
  
print(returnString())
```

The colon string indicates a string will be returned from the function. The bottom line `print(returnString())` calls the function and then prints the return value from the function which is a string returned.

The following would NOT parse correctly because indicated return value of type int is not being returned from the function.

```
function returnString() : int {  
    return "a string returned"  
}  
  
print(returnString())
```

Will NOT parse and will cause an Incompatible Types error.

Syntax Error Statements:

Catscript also has error handling for statements like the error handling for expressions. These errors Unexpected Tokens and Incompatible Types errors. Below are a couple examples of errors Catscript can catch.

```
var x = 10  
x = "string"  
print(x)
```

The data type of x became an int on the first line. Trying to reassign the type to string will cause a parse error of Incompatible Types.

Parse Errors Occurred:

Line 4:x = "string"

^

Error: Incompatible types

In the following case, the variable y is never assigned a value before trying to reference it. This will be caught by the parser with an error message that the symbol is not defined.

```
print(y)
```

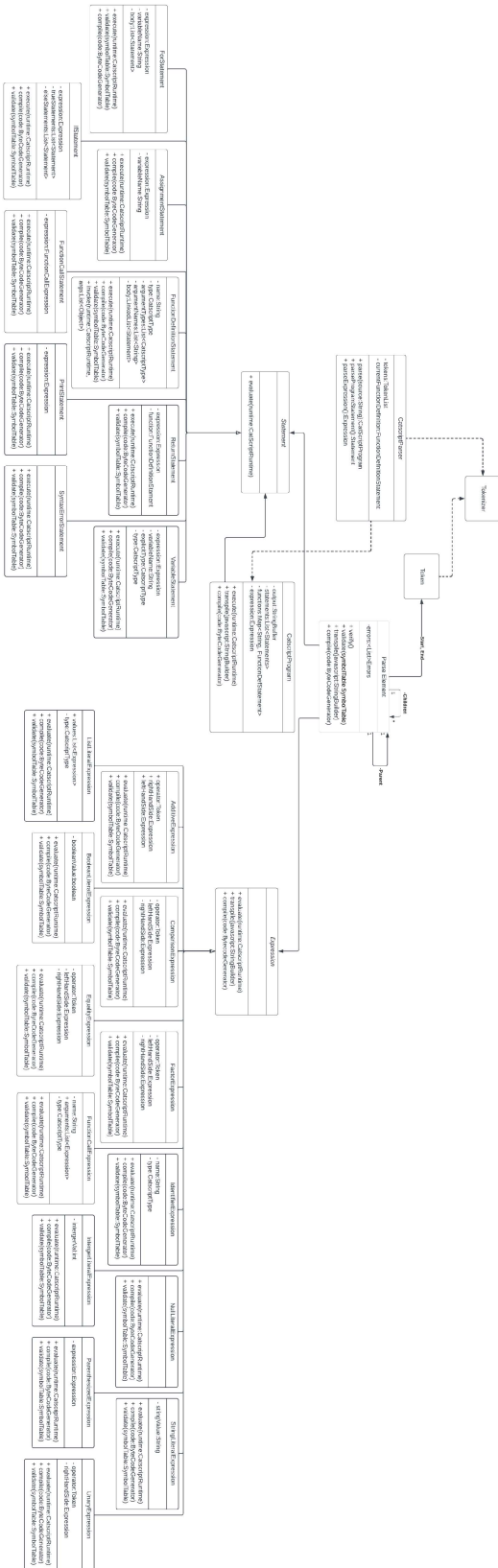
Parse Errors Occurred:

Line 3:print(y)

^

Error: This symbol is not defined

Section 5 – UML: Will have to zoom to see fully



Section 6 – Design Tradeoffs:

One of the design tradeoffs that were made for this project was to use a recursive descent parse that was hand coded over a parser generator. The reason that recursive descent was chosen is because it was easier to understand and allowed for the developer to find tune the parser in the code and the dev could have more control over the parser. Also the use of regular expressions was going to be overly complicated and it would be easier to write the parser by hand. Writing the parser by hand also allowed for the code to be much more easy to debug and the code would be much shorter than it would be with a parser generator.

Another design trade off was the use of abstract classes over interfaces. The reason that we used classes is because it would be easier to make test classes. We also used classes because it allowed for errors to be printed such as class needs to be implemented and this allowed to it easy to find things wrong with the code. Having these errors printed make it easier to learn how to do the parser.

Section 7 – Development Lifecycle:

The test-driven development cycle that we used for the project is test driven development. Meaning that tests for the project were written before the code was written. This was really helpful because it gave the developer goals to reach for and gave very clear guidelines to what needed to be done. It also allowed for the developer to account for many different things that could happen when Catscript code is given to the tokenizer, parser, and compilers. The test-driven development was also helpful because it allowed for testing of individual parts of the project made the development of each piece much easier.