

**Montana State University**

**CSCI 468 Capstone Portfolio – Catscript Compiler**

**Christopher Danielson**

**Brandon Marceau**

**Spring 2022**

**Professor – Carson Gross (Sponsored by JetBrains)**

## Section 1: Program

The source code for the Catscript Compiler is located in source.zip located in this directory ([csci-468-spring2022-private/capstone/portfolio/source.zip](https://github.com/csci-468-spring2022-private/capstone/portfolio/source.zip))

Source code was written in Java using the JetBrains IntelliJ IDE.

## **Section 2: Teamwork**

### **Team Member 1:**

Team member 1 was responsible for keeping the Github repository up to date and writing the bulk of the code for the Catscript compiler. Most of the time spent on the project was spent on writing Java code for tokening, parsing, and compiling within the Catscript programming language. They were also responsible for ensuring that all tests within the test environment were passing to complete checkpoints before the due date.

### **Team Member 2:**

Team member 2 was responsible for providing some unit tests for team member 1 to use in testing on certain aspects of Catscript to ensure they were working properly. Team member 2 also provided technical documentation for Catscript which can be found in section 4 of this document.

### **Time Estimates: Total Time 210 Hours**

Team Member 1 contributions: Code, Github, Checkpoint

Team Member 1 time: 200 hours (95.2%)

Team Member 2 contributions: Unit tests, technical documentation

Team Member 2 time: 10 hours (4.8%)

## Section 3: Design Pattern

### The Memoization Pattern:

The memoization pattern is used as an optimization technique for expensive function calls. The result of certain function calls can be stored in a cache using a hash map. When the function is called again, the result that is stored in the map can then be returned rather than having the entire function run through again. This optimizes programs when the same function call is being made many times by saving some computation time.

### Occurrence in Catscript:

The memoization pattern occurs at `src/main/java/edu/montana/csci/csci468/parser` in the `CatscriptParser.java` file. Following is the code that performs the memoization pattern with comments for explanation of each line. Note that this is not a thread safe implementation.

```
// Start by creating a map to store the CatscriptType
static HashMap<CatscriptType, ListType> cache = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {

    // Get the type if one already exists
    ListType listType = cache.get(type);

    // If the list type is null, assign the type and store in the map
    if (listType == null) {
        listType = new ListType(type);
        cache.put(type, listType);
    }
    return listType;
}
```

## Section 4: Technical Writing – Catscript Documentation

### Introduction:

Catscript is a statically typed programming language, meaning that when a type I declared the variable will stay that way for the execution of the program. Catscript has six different types of variables that include Ints, Strings, Booleans, Objects, Lists, and Nulls. There is also a variety of statements that can be written in Catscript, and these include For loops, If statements, Print statements, and Function declaration. Along with these different statements there is an array of expressions that can be written in Catscript. With these different things Catscript is a very basic programming language that allows for the most basic statements and expressions.

### **Expressions:**

Expressions in Catscript evaluate to some value. Expressions can be a literal value or they can be an expression that evaluate to a literal value. These literal values are represented by Integers, Strings, Booleans, Nulls, Objects, or a List.

### **Integer Literal Expression:**

Integer literal expressions in Catscript are just a 32-bit number. Integers can be implemented by just typing a number in Catscript.

Example:

```
1
```

This will evaluate to 1

### **String Literal Expression:**

String Literals in Catscript is represented by a word, phrase, or sentence that is incased in double quotes. The area inside the quotes is not case sensitive as well.

Examples:

```
"Hello World"  
"Brandon is Cool"
```

These two String will evaluate to Hello World and Brandon is Cool

### **Boolean Literal Expression:**

Boolean literal expression is a true or false expression in Catscript

Examples:

```
true  
false
```

The first value will evaluate to true and the second one will evaluate to false

### **Null Literal Expression:**

Null literals are a type in Catscript that represent an empty value and will return as nothing.

Example:

```
null
```

This value in Catscript twill evaluate to null

### **Object Literals:**

Objects in Catscript are used to store the other types of values like Strings, Integers, Booleans, and Nulls. The only difference that these values will have from a String is that they are an object type. Objects can only be declared along with a variable.

## List Literal Expressions:

Lists in Catscript are a list of values that can be stored. The list has to be one type of values when you store values. So, you can have a list of Integers or a list of Strings but there can also be a list of objects as well. So, there can be a list that contains a null, String, Integer, and Boolean all at the same time but they have to be of type object. There can also be lists inside of lists, and there can be a list of lists.

Examples:

```
[1, 2, 3]
[1, null, true, "Brandon is Cool"]
[1, 2, ["Brandon", "is", "Cool"]]
```

These are all valid examples of lists in Catscript

## Unary Expression:

Unary expressions are used to make values in Catscript negative or to make a true a false or a false a true. The way that you make a Integer negative in Catscript is by adding a “-“ in front of it. The way that a Boolean can be reversed in Catscript is by adding a “not” in front of the value.

Examples:

```
-1
-43
not true
not false
```

The two numbers will evaluate to -1 and -43 and the “not true” will equal false and “not false” will equal true.

## Equality Expression:

Equality expressions are used to compare two values and then evaluate to true or false depending on the comparison. You can compare Integers, Strings, Nulls, and Objects in Catscript to return a Boolean value. The `==` will check if the values are equal and if they are it will return true otherwise false. The `!=` will check if the values are not equal and if they are it will return true otherwise false.

Example:

```
"Bandon is" != "Cool"  
1 == 1
```

The top will evaluate to true because they are not equal. The bottom will be true because 1 is equal to 1.

## Comparison expression:

Comparison is used to compare two values and see if they are greater than, greater then or equal, less than, or less than or equal. The expression will return a true or false value and can be used to compare Integers. The `>=` will check if the left-hand value is greater than or equal to the right side and will return true if its greater than or equal otherwise it will return false. The `>` will check if the left side is greater than the right side and return true if it's greater than and false otherwise. The `<=` will check if the left side is less than or equal to the right side and will return true if left is less than or equal to right and false otherwise. The `<` will check if the left is less than the right side and return true if left is less than right and false otherwise.



Examples:

```
1 < 4  
1 <= 1  
5 > 3  
5 >= 5
```

The top will evaluate to true

The second one will evaluate to true

The third one will evaluate to true

The fourth one will evaluate to true

### **Additive Expression:**

Additive expressions are used to do addition and subtraction with Integers. The + is used to add Integers together and – is used to subtract Integers.

Examples:

```
1 + 1  
2 - 1
```

The top one evaluates to 2 and the bottom one evaluates to 1

### **Factor Expression:**

Factor expressions are used to do multiplication and division in Catscript between Integers. The \* is used to do multiplication and / is used to do division.

Examples:

```
4 / 2  
2 * 2
```

The top will evaluate to 2 and bottom will evaluate to 4

### **Function Call Expression:**

Function call expressions are used to call functions that have been declared and they could return any type to a variable. It could also return void and not have a return statement. Parameters can also be passed the function so they can be used in the local scope of the function. These parameters are optional if the function was declared without parameters.

Examples:

```
x = foo(3)
foo(4)
foo()
```

above are three examples of function calls in Catscript

### **Syntax Error Expression:**

When an error is found in the code an error is added onto the parse tree with a type of `SyntaxError`. The parser will not stop until things are evaluated when syntax errors will be thrown.

### **Statements:**

#### **Variable Statement:**

Variable statements are used to declare variables in Catscript. There is more than one way that the variable statement can be used. The first way that it can be used is by declaring the variable with a type associated with the variable. The way that this is done is after the name of the var is give the type can be specified but this is optional. The second way that a variable can be declared is by just giving the name and the value that goes with the variable.

Examples:

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

These are three ways to declare variables in Catscript

### **Assignment Statement:**

Assignment statements are used to reassign variables in Catscript. The way that the variables are reassigned is by calling the variable with an equal sign and a new value. There is a type system so the type of the variable must match with the new value.

Example:

```
x = 10
```

the var x is being reassigned to be 10

### **For Statements:**

For statements are the loop for Catscript. For loops in Catscript are a loop that executes a block of statements that is in the for-loop body.

Example:

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

above is an example of a for statement in Catscript

### **Function Definition Statement:**

The function definition statement is used to make a function in Catscript and can be called later in the program. There can be parameters that are passed to the function and types can be given to these parameters. Giving a type on the parameter is optional. Functions are also not

required to have a return statement in them, and the type of the return isn't required to be defined. Inside the function body there is a list of statements that are executed when the function is called

Example:

```
function x(a : object, b : int, c : bool) {}  
function x(a, b, c) {}
```

Above are two examples of how to declare a function

### **If Statement:**

If statements in Catscript are used to execute a set of statements based on how a expression evaluates. The expression that is in the if statement must evaluate to a Boolean and based on that value the if statement or the else statements will execute. The else statements are statements that will run if the expression is false. The else statements are optional for the if statement but have an else requires there to be an if beforehand.

Example:

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

Above is an example of two if statements

### **Print Statement:**

Print statements in Catscript are used to print a value to the terminal. The print statement takes either a variable or value as input and can print them to the terminal. The print can handle all types and can print all of them.

Examples:

```
print(1)  
print(x)
```

Above is a print handling both variables and values

### **Return Statement:**

Return statements are used in functions to leave the function with a value that can be stored in a variable at a function call. There can be more than one return statement in a function but they all have to be reachable so there can't be two in a row because the second one will never get reached. Returns can return both variables and values and can return all types in Catscript.

Example:

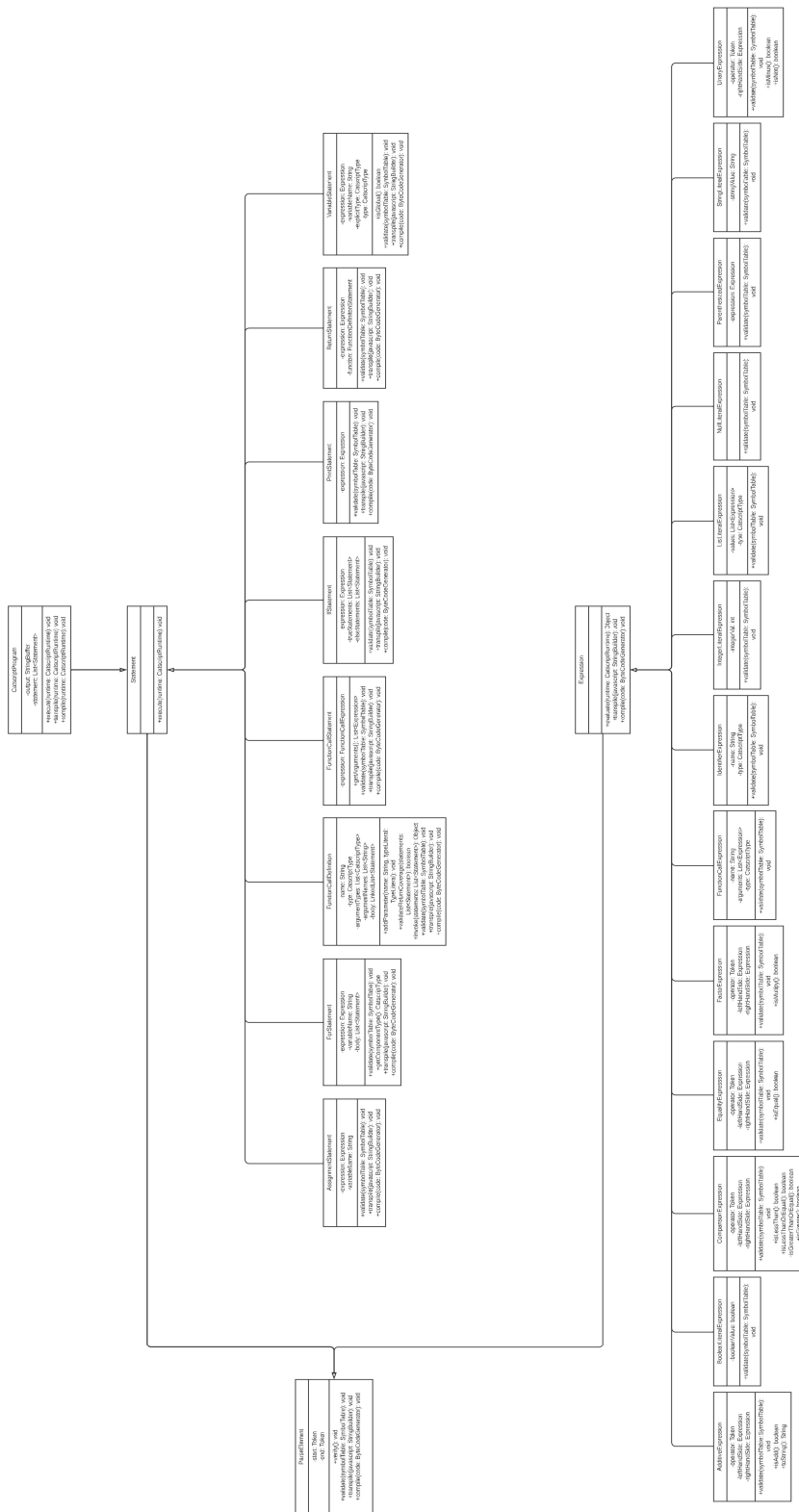
```
return 1  
return x
```

example of return statements in Catscript

### **Syntax Error Statement:**

When a syntax error is found in the statements an error is added to the parse tree and then the program will continue to parse. The error will be thrown right before the parse trees are evaluated so all errors can be found before one is thrown.

## Section 5: UML



## **Section 6: Design Trade-offs**

### **Main Trade-Off:**

The main trade-off for Catscript was the use of a Recursive Descent parsing algorithm rather than using a Parse Generator. Each of these methods of parsing will be described in further detail below, as well as why Recursive Descent was chosen over a Parse Generator.

### **Parse Generator:**

Parse Generators are composed of two parts a lexical grammar and a long grammar. ANTLR is an example of a popular and widely used Parse Generator. The lexical grammar is in the form of REGEX (regular expressions), and the long grammar is in the form of BNF (Braccus Naur Form) or EBNF (Extended Braccus Naur Form). The parser in a Parse Generator produces a syntax tree, and it does perform recursive descent. One of the advantages of Parse Generators is that starting infrastructure is easier to set up and can require less code. Parse Generators are harder to read, and it can be more difficult to learn how to use a Parse Generator due to some abstractness of the recursive descent working. Since it can be more difficult to read and gain a deep understanding of, Parse Generators were decided against.

### **Recursive Descent:**

Recursive descent is a top-down parsing technique. A parse tree is built from the top down creating branches for each production of code until every branch ends in a terminal (a literal value). Using recursive descent is much less abstract since the result of the parse very closely mimics the grammar that is followed while building the parser. Every method that can be called to build the parse tree can be seen on the right-hand side of the grammar. Since a programmer can understand and read how the tree is built, it makes this method of parsing more intuitive for less experienced programmers and helps them learn more about how languages are

built. It is an efficient way to design a compiler, and it does not require the use of overly complex design patterns. Due to these, this method of parsing was chosen for the Catscript project.



## Section 7: Software Development Life Cycle Model

### Test-Driven Development:

Test-Driven Development (TTD) is a development model that uses sets of tests that will test some aspects of the program. The tests will fail originally, and a programmer's goal is to write code simple enough to make each test pass. Once a test passes, the programmer can focus on the next test, eventually creating a large piece of software. The advantages of TTD can include helping developers focus on small pieces and doing those pieces well, and a cleaner more readable code base. TTD was used in the development of Catscript with the Professor providing large sets of tests that needed to be passed by a certain due date. There were 4 sections of tests for the student developers to complete throughout the semester. The first set of tests was for testing that tokenizing was working properly. The second set of tests was to ensure that the parser was working. Third, was a set of tests that checked the expressions and statements evaluated properly. The last set of tests was checking to see if the code was properly compiled to JVM bytecode. Each set of tests needed to be completed before moving on to the next set of tests.

Test-Driven Development was very well received by the student developers. It is a very good way to be able to check whether small pieces of code are working properly as they are written. It allows a developer to focus on creating small pieces of functionality and not worry as much about the overall big picture. The big picture will come together when all the small pieces have been completed. By having tests with expected outputs, the developer could also more easily debug code that was not working as expected. It also gave student developers a place to start from and know exactly what they should work on next. It really helped keep each developer on track and slowly working through parts to complete a whole project.