

# **CSCI468: Compilers**

Mark Braun, Grant Dunbar

Spring 2023

# Section I: Program

A compressed .zip file of my entire source directory is located at:

<https://github.com/markrb19/csci-468-spring2023-private/tree/main/capstone/portfolio>

## Section II: Teamwork

Work on this project was completed solely by team member 1 and team member 2. Team member 1 and team member 2 individually completed individual codebases for CatScript over the course of the semester. Team member 1 and team member 2 each wrote three code-based tests (found in *src/test/java/edu/montana/csci/csci468/demo/Scratch.java*) for each other designed to evaluate the accuracy of each other's code. Additionally, team member 1 and team member 2 each completed the technical documentation of each other's CatScript implementation. Working with a partner over the course of this project was immensely helpful to overcoming roadblocks and sharing unconventional or remarkable implementations that both team members were struggling with. Both team members had good understandings of what each other consistently struggled with and were able to push each other with the code tests that we developed for each other to excel in these areas.

## Section III: Design Pattern

In my implementation of my capstone project this semester, one design pattern that was used was memoization. This design pattern can be found beginning on line 36 of the java file *CatscriptType.java*, found in the file path:

*src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java*

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

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

I used a memoization design pattern for this component of my capstone project to keep my implementation memory-efficient and avoid creating new list types redundantly with each encountered component type. With this design pattern, the first time each CatScript component type is used in a list type variable, the mapping from this component type to the desired list type will be stored in a hash map, making it easy to reference for subsequent list variable initializations.

The main advantage of using memoization in this way is that calling the *getListType()* function with previously encountered Catscript types as parameters will result in returning a result much more quickly than if the Catscript list type had to be reevaluated on each individual function call. The main downside of using a memoization pattern like this is that there is more time and space overhead for early calls to this function. Additionally, if Catscript types are not used in lists multiple times in a Catscript program, the benefit of the pattern likely won't be fully realized. However, in Catscript programs with numerous distinct lists with the same component type, or numerous calls to *getListType()* with the same type parameter, this pattern will greatly speed up this function call.

## **Section IV: Technical Writing**

The technical document team member 2 completed documenting CatScript will begin on the following page.

# Catscript Guide

---

This document is written to satisfy capstone requirement 4.

## Introduction

---

Catscript is a simple scripting language. It is a statically typed functional programming language that is built using Java. Catscript includes many features as well as variable types that give you a broad range of use for this language.

## Features

---

### For loops

Catscript allows the use of for loops. For loops are a type of iterator that allow you to iterate over a list and use the value of the string as you iterate over it.

#### Structure

```
'for', '(', IDENTIFIER, 'in', expression ')', '{', { statement }, '}'
```

#### Example

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

### If Statement and Else Statement

Catscript allows the use of If Statements as well as Else Statements. If statements allow you to have different operation ran if a condition is either met or not met.

#### Structure

```
'if', '(', expression, ')', '{', { statement }, '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ]
```

#### Example

```
if(x == 1){  
  print("Odd")  
} else if (x == 2){  
  print("Even")  
} else {  
  print("Too big to compute")  
}
```

### Print Statement

Catscript has a built in print function that allows you to output expressions.

#### Structure

```
'print', '(', expression, ')'
```

#### Example

```
print("Catscript is the best programming language!")
```

### Variable Statement

Catscript allows you to define that can be used on a local or global level of your program. These variables can be of type int, string, boolean, list, or object. The specific types will be outlined later in this document.

## Structure

```
'var', IDENTIFIER, [':', type_expression, ] '=', expression
```

## Example

```
var z:int = 5
```

## Assignment Statement

Catscript allows a programmer to redefine an already defined variable. This feature allows for updates to be made to a variable without having to create a new variable name. The type of this variable does matter. You can not reassign a typed int variable as a string, but you can reassign a typed object that is storing an int to be updated to storing a string.

## Structure

```
IDENTIFIER, '=', expression
```

## Example

```
var z:int = 4  
z = 5
```

## Return Statement

Catscript allows for functions to use a return statement. This return statement can either be used to return void or it can return an expression that can be used in another function. This allows a programmer to create fruitful functions as well as void functions.

## Structure

```
'return' [, expression]
```

## Example

```
if (x >= 5) {  
    return false  
}
```

## Function Definition

Catscript allows for functions to be defined. This allows a programmer to create repeatable code without needing to copy and paste the code over and over to use the same code blocks. Functions allow for arguments to be passed in and for an expression to be returned at the end of the function using a return statement.

## Structure

```
'function', IDENTIFIER, '(', parameter_list, ')' + [ ':' + type_expression ], '{', { function_body_statement }, '}'
```

## Example

```
function foo(x:int, y) { print(x*y) }
```

## Function Call

Catscript then allows for functions to be called after they have been declared. When a function is called, arguments defined in the function definition statement are required and if a function has a return value, this can then be stored in a variable.

## Structure

```
IDENTIFIER, '(', argument_list , ')'
```

## Example

```
function foo(x:int, y) { print(x*y) }  
foo(20, 23)
```

## Equality and Equality Expressions

Catscript allows for equality and inequality checks as well as comparison checks.

### Structure

```
expression == expression  
expression != expression  
expression >= expression  
expression > expression  
expression <= expression  
expression < expression
```

## Example

```
if (x == 5) {  
    return true  
}  
if (y != 4) {  
    return false  
}  
if (x > 5) {  
    return "Greater than"  
} else if (x >= 4) {  
    return "Greater than or equal"  
} else if (x < 4) {  
    return "Less than"  
} else {  
    return "Less than or equal"  
}
```

## Math Expressions

Catscript allows for the use of mathematical operations. Recursive descent is used in these math expressions to show the order of operations. Addition can be used for both integer addition as well as string concatenation, but the other mathematical operations can only be used on integer variables and inferred int type variables.

### Structure

```
expression, "+", expression  
expression, "-", expression  
expression, "*", expression  
expression, "/", expression
```

## Example

```
var x:int = (5 + 4)/3 * 2
```

## Unary Expression

Catscript allows for unary expressions which lets an expression be negated. The "not" or "!" can be used to flip a boolean value from either true to false, or false to true while a "-" will negate an integer value making a positive number negative and a negative number positive.

### Structure



```
"not", expression
"!", expression
"-", expression
```

## Example

```
var x = false
if (!x) {
    print("true")
} else {
    print("false")
}
```

## Types

### Integer

Catscript has an integer type built in to it that allows for variables to be defined as a whole integer value.

#### Example

```
var x:int = 468
```

### String

Catscript has a string type built in to it that allows for variables to be defined as a string of characters.

#### Example

```
var x:string = "Catscript"
```

### Boolean

Catscript has a boolean type built in to it that allows for variables to be defined as either true or false.

#### Example

```
var x:bool = false
```

### Object

Catscript has an object type built in to it that allows for variables to be able to store a value of any of the other types. Object is the inferred type and is used if a type is not defined. Object type variables can be defined as an object by either passing in the object type or omitting the type declaration. If a variable or argument is defined with a bad type, such as boool instead of bool, object type will be used for the variable or argument.

#### Example

```
var x = 5
var y:object = "This is an object"
```

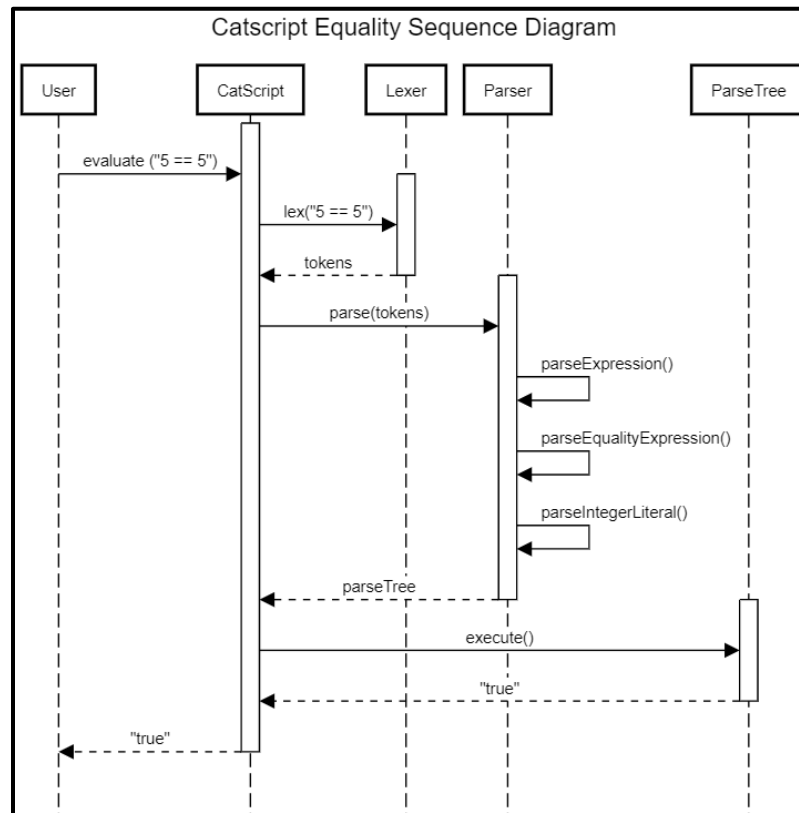
### List

Catscript has a list type built in to it that allows a variable to contain more than one value. This list can have a specific type that will require all values to be of that type and also has an object type that will allow any value to be stored in the list.

#### Example

```
var array:list<int> = [1,2,3,4]
```

## Section V: UML



This UML diagram is a sequence diagram of the parsing and execution of an equality expression in Catscript. Here, the equality expression being parsed is the Catscript expression "5 == 5". First, Catscript sends the expression to the lexer, which divides the entire expression up into its component tokens (here, those tokens would be ["5", "==", "5"]). The lexer returns these tokens to the Catscript program, which then sends the tokens to the parser by way of a *parse()* function call. Then, our recursive descent parser begins parsing these tokens, beginning at the top level with a *parseExpression()* function call, which recursively works its way down the Catscript hierarchy of expressions, based on the Catscript grammar. This recursive descent ends at the *parseEqualityExpression()* function call, where an appropriate operator ("==") is identified. Finally, the parser parses the integer literals on either side of the equality expression and returns the complete parse tree to the Catscript program. Then, after an *execute()* function call on the parse tree, the equality expression is evaluated for truth, the return value of which is returned first to the Catscript program and subsequently to the user.

## Section VI: Design Tradeoffs

One of the major design choices we made in the creation of this Catscript compiler was the choice to create a recursive descent parser, instead of an alternative parsing style like a parser generator. There are several substantial advantages of using recursive descent in this situation. The first major advantage of this style of parser is that is relatively easy to impose our own “order of operations” onto the parsing of expressions and statements. This is true in the traditional “multiplication happens before addition” sense but is generalizable to the enforcement of a complete hierarchy of parsing precedence. With recursive descent, we have complete control over the order of the recursive calls that parse each subsequent type of expression. The second major advantage of using recursive descent parsing as we have is that there is no need to learn a separate language of a parser generator – we were able to write the recursive descent parser in Java, which was a language we were already comfortable with, which reduced the time and effort overhead to complete this project.

There are several disadvantages, or tradeoffs, that we had to concede when we decided to use a recursive descent parser. The first disadvantage to consider for this parser is it’s efficiency – generating sequences of recursive calls for each expression that needs to be parsed can quickly become cumbersome and not as fast as might be desired. Additionally, a technique like parser generators is somewhat industry standard in today’s day and age, so there might be a lack of practical experience gained from this. Additionally, there is much more overhead in terms of manual implementation using this method than there is associated with a parser generator, which automates away much of the explicit specification of a recursive descent parser. Finally, recursive descent parsing is generally less flexible with different types of grammars – it didn’t matter for this project since the Catscript grammar was very simple and compatible with recursive descent, but left-recursive and ambiguous grammars can be harder to parse with this method.

# Section VII: Software Development Life Cycle Model

For our capstone project, we used test driven development. There were several major checkpoints over the course of the semester (the tokenizer, the parser, the compiler), that each had test suites associated with them. These test suites were comprehensive and ensured that the complete functionality of each project component was correct. This model was very helpful during the implementation process, and the subsequent debugging of issues. Each test was very compartmentalized, and only evaluated specific attributes of each feature of Catscript. This made it very clear where errors were arising, which inputs caused the errors, and specifically which cases were not being handled. This was helped greatly by the fact that the tests were transparently available to us to view. There was no mystery in what needed to be accomplished, and we could see, side by side, the input to the test, the expected output, and the output that our implementation created. This also made helping each other within our team with implementation very easy, as we were able to write tests for each other to test certain components of functionality that we thought each other might struggle with.

There were a couple of disadvantages to test driven development. The first disadvantage of this method was that it is very tempting to write implementation that technically passes provided tests but is not necessarily the best implementation. This does not mean that implementations were specifically hardcoded to pass tests, but more that there were a finite number of tests that tested a finite number of components of functionality, and there were necessarily areas of functionality not tested by the test suites. It was important, but difficult, over the course of the project to make sure that full functionality was available, and not merely the minimum functionality required to complete test suites. The second disadvantage of this process was the pure volume of tests that were required to evaluate a code base as large as this compiler ended up being. There was close to 200 tests that were necessary to evaluate functionality, and this became very cumbersome very quickly, as well as making identifying redundant tests and information difficult.