

# Capstone Portfolio

CSCI-468 Compilers  
Spring 2022

River Kelly

Tester: Kyler Gappa

# Program

The complete project source code can be found at:

<https://github.com/RK-MSU/csci-468-spring2022-private/blob/master/capstone/portfolio/source.zip>

## Teamwork

The workload for this project was completed in five distinct stretches; Tokenization, Parsing, Evaluation, Bytecode generation, and Partner Testing. Each individual partner created their own solutions to the Catscript scripting language implementation. Then, created tests for each other. The code bases each individual partner created had to withstand these tests that we shared with one another.

<b>Member</b>	River Kelly	Kyler Gappa
<b>Role</b>	Primary Developer	Software Test and Technical Documentation
<b>Estimated Hours</b>	115	15

**Total Estimated Hours:** 130

## Design Pattern

One of the design patterns used in the Catscript scripting language was Memoization, also known as Flyweight. The design pattern is used to help assist in reducing the redundancy of an operation. The core principle of this design pattern is such that the results of a function or process should be remembered. Let's say that a function is invoked many times during the lifetime of an application, and the functionality of this method is repeated identically. Instead of computing the operations each time the method is called, the method stores the input and saves the corresponding result. This way, the next time the method is called with the same input, the result is already stored and no further processing is needed, the result may be returned immediately.

The usage of the Memoization design pattern can be seen in the Catscript codebase within the method `getListType()` under the class *CatscriptType*. By implementing this design pattern, only one *ListType* is created for each *CatscriptType*. Doing so improves the runtime speed and reduces the overall memory cost by eliminating duplicate object creation.

The location of this design pattern can be found at:

<https://github.com/RK-MSU/csci-468-spring2022-private/blob/master/src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java#L35-L52>

```
// cache storage for ListTypes
static final HashMap<CatscriptType, ListType> LIST_TYPE_CACHE = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    // get the list type from cache storage
    ListType list_type = LIST_TYPE_CACHE.get(type);

    // check if list_type exists
    if (list_type == null) { // list_type does NOT exist
        // create a new instance of the ListType
        list_type = new ListType(type);
        // insert the new list type instance into our cache storage
        LIST_TYPE_CACHE.put(type, list_type);
    }

    // return the list_type
    return list_type;
}
```

# Technical Writing (Catscript Guide)

## Introduction

Catscript is a small statically typed scripting language that compiles to JVM bytecode featuring both evaluation and compilation. It is designed to take the best features of Java and combine it with features inspired from other languages to create an ideal language to work with.

Here are some features:

## Features

## Comments

Comments are used in Catscript using the `"/"` for single-line comments and `"/ * */` for block comments

```
/*
 *This is
 a multiline comment
 */
var x = 8
//This is a single line comment
print(x)
```

## Types

Catscript supports basic types for variables. These types can be both explicitly declared and inferred. Catscript also shares namespaces, variable names from all fields and sub-scopes are stored in the same location. Here is the small list of Catscript's type system:

- int - a 32 bit integer
- null - a null type
- bool - boolean value
- string - Java string object
- list - a list of object
- object - any value

## For Loops

Using the 'in' keyword like most other languages, for loops in Catscript are used to iterate through contents in an array as shown here:

```
var list = ["a", "b", "c", "d"]
for(i in list){
  print(i)
}
```

This returns "abcd". i is assigned to each character within the array and printed as the loop iterates through the array.

## If Statements

If statements in Catscript are identical to Java. Using the if, else if, and else format, the user uses boolean literals or expressions that evaluate a boolean as well as comparison operators to evaluate if statements.

```
var x = "yes"
if(x == "yes"){
  print("accepted")
}
else if(x == "no"){
  print("denied")
} else {
  print("who are you")
}
```

The above returns the string "accepted".

## Function Definitions

Functions are defined by including the keyword "function" before your identifier and parameters. The return type is void, unless otherwise stated.

### Explicitly Typed

This function uses explicit types for the parameters and returns an int

```
function foo(num: int, truth: bool) : int {
  //function operation
  return num
}
```

### Type Inferred

This function has a default return type of void. It also uses type inference for the arguments.

```
function foo(num, truth){
  // function operation
}
```

```
}
```

## Unary Expressions

Catscript uses the negate (-) unary operator on integers and booleans take the "not" keyword to negate its value

```
var x = 20
print(-x) // This will print "-20"
return not true // This returns false
```

## Comparison

Catscript uses the basic comparison operators less than, less than or equal, greater, greater than or equal, as shown below:

```
1 < 0 // false
1 <= 0 // false
1 > 0 // true
1 >= 0 // true
```

## Equality

Catscript also check for equality using the basic equality operators on objects of the same type. The operands used are "equal equal" and "exclamation point equal"

```
1 == 0 // false
1 != 0 // true
```

## Variable Statements

Variables are declared using the "var" keyword

## Explicitly Typed

You can explicitly define an object's type by including a ":" after var but before your "="

```
var num: int = 8
var hello: string = "Hello World!"
var truth: bool = false
var nums: list<int> = [1,2,3,4,5]
```

## Inferred Type

If you don't explicitly give your variable a type, the type will be inferred while the statement is parsed

```
var num = 8 //int
var hello = "Hello World!" //string
var truth = false //bool
var nums = [1,2,3,4,5] //list of type int
```

Catscript also allows for lists of type object with multiple dimensions:

```
var list = [8, "Hello World!", [8, 40, 320]]
```

## Print Statements

Catscript print statements are similar to those in other languages where the keyword "print" is used to return data to the user.

```
var num = 8
print(num) //prints out "8"
print("Hello World!") //prints out "Hello World!"
```

You can also use the "+" operator to concatenate integers to your strings

```
var num = 1
print("Hello player " + num)
//returns "Hello player 1"
```

## Math Operators

Catscript uses basic division and multiplication operators to calculate integers

```
var x = 10
var y = 40
print(x*y) // returns 400
print(x/2) // returns 200
```

Catscript also uses basic addition and subtraction operators

```
print(8+40) // evaluates to 48
print(8-40) // evaluates to -32
```

## Catscript Grammar

```
catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            function_call_statement;

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
               '{', { statement }, '}'

if_statement = 'if', '(', expression, ')', '{',
              { statement },
```



```

        '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];

print_statement = 'print', '(', expression, ')'

variable_statement = 'var', IDENTIFIER, [ ':', type_expression, ] '=', expression;

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

function_declaration = 'function', IDENTIFIER,
    '(', parameter_list, ')' + [ ':' + type_expression ],
    '{', {function_body_statement},'>';

function_body_statement = statement |
    return_statement;

parameter_list = [ parameter, {',' parameter } ];

parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [ , expression];

expression = equality_expression;

equality_expression = comparison_expression { ("!=" | "==")
    comparison_expression };

comparison_expression = additive_expression { ">" | ">=" | "<" | "<=" )
    additive_expression };

additive_expression = factor_expression { ("+" | "-" ) factor_expression };

factor_expression = unary_expression { ("/" | "*" ) unary_expression };

unary_expression = ( "not" | "-" ) unary_expression | primary_expression;

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |
    list_literal | function_call | "(", expression, ")"

list_literal = '[', expression, { ',', expression } ']';

function_call = IDENTIFIER, '(', argument_list , ')'

argument_list = [ expression , { ',', expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list'

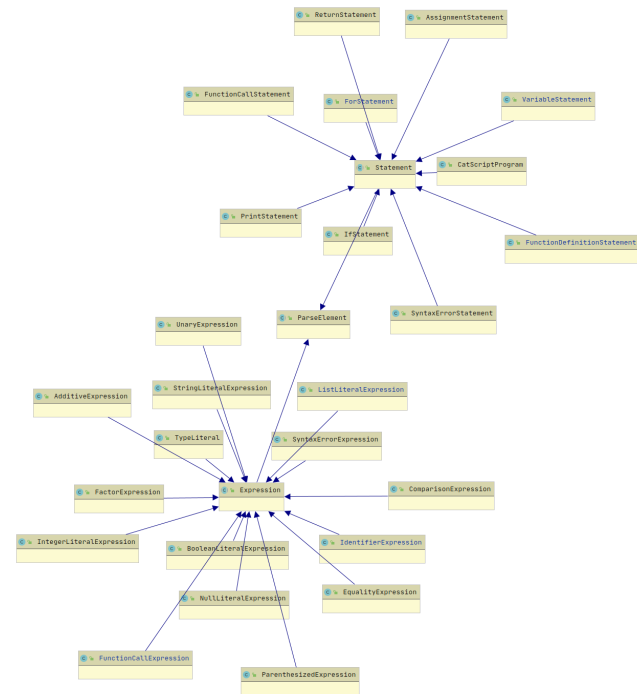
```

```
[, '<' , type_expression, '>']
```

## UML

Although there was no required UML creation required for this project, the UML diagram to the right explains how each of the components within the Catscript systems are related.

The *ParseElemnts* within the Catscript language are divided into two categories; Statements, or Expressions. All of the child-most parse elements fit into one of these types.



## Design Trade-Offs

There was two primary design trade-off for this project. The first design trade-off for the project was the use of Recursive Descent v.s. Parser Generator. The second design trade-off for this project was the specificity of separation of concerns.

Recursive descent was chosen for this project because this implementation style more closely aligns with how the grammar of the language is interpreted. Although this design trade-off, Recursive Descent, requires more lengthy and written-out routine procedures, its overarching functionality is easier to comprehend.

When creating a solution to a software problem it is highly valuable to consider the unique components within the complete application. It is commonly favorable in software engineering to “tightly couple” certain procedures within a software application. This process is referred to as the separation of concerns. This technique provides modularity and defined scopes for which a component operates/interacts with the system. For this project, the separation of concerns was less desirable compared to simplicity and code location. Evaluation and compilation were directly connected to the parse tree nodes. This is unusual because these two distinctly unique procedures would typically be separated (i.e. they have different concerns). This trade-off was made to enhance simplicity throughout the makeup of components within the software system.

# Software Development Lifecycle Model

The Catscript scripting language project was developed using test-driven milestones. Each phase of the project had a suite of Junit tests that determined the completion of each stage. The successful completion of each of these tests determined the integrity of the software system as a whole. The tests began with the first stage of the project, Tokenization, and continued to Parsing, Evaluation, and finally Bytecode generation. The successful completion of each step in the project was dependent on the previous steps. Without Tokenization, Parsing was not possible, and without Tokenization and Parsing, Evaluation was not possible.

The integrity of the project was maintained using Git version control. This allowed proper maintenance of the completion of each stage throughout the test-driven milestones.