

Montana State University

Caden Fong

Tester: Kaylee Fong

Semester: Spring 2022

Course Name/Number: Compilers / CSCI 468

## Section 1: Program

<https://drive.google.com/file/d/1GQH7O6r25QLMSZ8jthPOy4UOfPnT-3Jd/view?usp=sharing>

## Section 2: Teamwork

This team consisted of two team members, team member 1 and team member 2. Team member one was in charge of writing the compiler and this capstone document. Team member one was the primary creator for this project since this was mostly an individual project. Team member 2 was tasked with writing three tests to test the compiler's code and also writing the documentation for the catscript language. Team member two was the tester for this project and did their own capstone where they were the primary creator. Around 180 hours were spent on this project in total. Team member 1 spent over 150 hours on this capstone assignment. Team member two, the testing member, spent around 30 hours on this assignment.

**Total Project Time:** 180 hours

### **Team Member 1 (Primary Creator):**

**Primary contributions:** Writing the compiler and this capstone document.

**Percentage of Time Spent:** 85% (150/180 hours)

### **Team Member 2 (Tester):**

**Primary contributions:** Writing three tests in the "PartnerTests.java" in the test directory for the compiler and the documentation for catscript.

**Percentage of Time Spent:** 15% (30/180 hours)

## Section 3: Design Pattern

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

For the design pattern to be implemented in this project, I chose to use the flyweight design pattern. In the project directory “src/main/java/edu/montana/csci/csci468/parser”, the design pattern is implemented in the file “CatscriptType.java”. The code above is the implementation of the flyweight design pattern. This pattern was chosen for two design reasons. The first reason being memory efficiency, without this pattern every time this method is called a ListType object is created using up more space than necessary since there are only so many different ListTypes, many being used more than once. Also, given larger programs, the speed of compiling and interpretation becomes faster as later calls to the method only perform a search in a hashmap rather than creating a new object. Therefore, the flyweight design pattern is a better choice than just coding in the method.

## Section 4: Technical Writing

---

# Catscript Technical Documentation

---

## Introduction

Catscript: a programming language that operates as laid out in this documentation. The EBNF for this language is seen below:

```
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 , '>']

```

## Features

### Comments

Catscript comments can be written with two forward slashes that comments out the rest line they are on.

```
// This is a Catscript comment
```

### Variables

Variables in Catscript are values that can be of type list, int, bool, string, or null. Explicit type can be given or omitted.

```

var variable : string = "foo"
variable = "bar"
or
var variable = "foo"
variable = "bar"

```

## Lists

Lists are an immutable type in Catscript, and can be component wise list, int, bool, string, and null.

```
var catScriptList : list <string> = ["Cat", "Script"]  
or  
var catScriptList = [0, 1, 2]
```

## Mathematical Operators

Addition in Catscript uses int addition and is overloaded to allow string concatenation.

```
1 + 2  
or  
1 + "foo"  
or  
"foo" + "bar"
```

Subtraction is similar to addition for ints.

```
1 - 1
```

Multiplication and division using basic mathematical factoring.

```
1 * 1  
and  
1 / 1
```

Unary Operator for negating ints and bools.

```
-10  
and  
not false
```

## Print Function

The Catscript print function sends string output to a buffer that then will use the JVM out system to print to the standard output.

```
print("foo")  
print(true)
```

```
print([0, 1, 2])
```

## For Statements

Catscript uses an iterable list for looping functionality and can consist of many statements in the body, ending when there are no more items in the provided list.

```
for (i in ["Say", "Hello", "to", "CatScript"]) {  
  print(i)  
}
```

## Equality and Comparison

CatScript comparison has the following operators less than, greater than, less than or equal to, and greater than or equal to.

```
1 < 1  
1 > 1  
1 <= 1  
1 >= 1
```

Equality compares object value and reference.

```
var reference1 = ["foo", "bar"]  
var reference2 = reference1  
reference1 == reference2  
or  
1 == 1  
or  
true != true
```

## If Statements

Catscript supports conditional code blocking and allows for multiple else if linking as well as nesting if statements.

```
var foo = "bar"  
var i = 1  
if (foo == "bar") {  
  if (i == 1) {  
    print("bar 1")  
  } else {  
    print("bar" + i)  
  }  
}
```

```
} else if (i == 2) {  
    print("not bar 2")  
} else {  
    print(foo + i)  
}
```

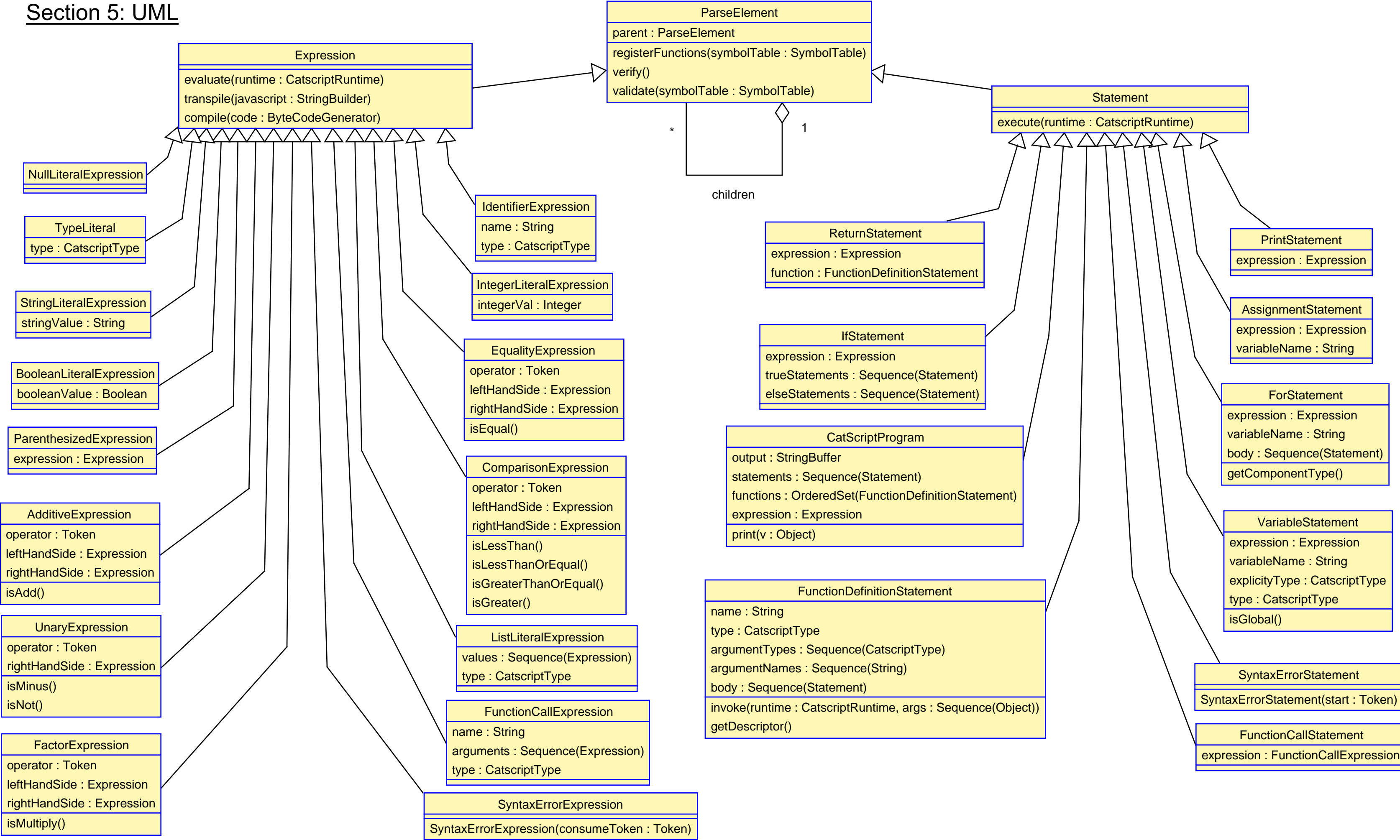
## Function Definitions and Calls

Catscript supports function definitions using unique identifiers and can use any amount of parameters in definition. Return type can be explicitly give or defaulted to void.

```
function foo(bar:bool):bool{  
    if (bar) {  
        return false  
    } else {  
        return bar  
    }  
}  
  
// Function Call Example  
foo(true)
```



Section 5: UML



## Section 6: Design Trade-offs

A design decision made at the beginning of this project was deciding the implementation method of the parser. The recursive descent algorithm was chosen for modeling my parser rather than a parser generator for a few different design reasons. First and foremost, one of the benefits of choosing recursive descent over a parser generator is the more simplistic implementation. Simply put, recursive descent is easier to implement because it is more concise when modeling a grammar as opposed to a parser generator, this is due to the fact that recursive descent is modeled more closely after the recursive nature of grammars. However, despite the simplicity, recursive descent requires more physical effort in creating a parser versus a parser generator, as it requires a more granular implementation of different parts. Another trade-off in choosing recursive descent is the education factor as recursive descent gives a more intuitive understanding of grammars rather than a parser generator's abstracted implementation. In conclusion, the best design trade-off decision for this project, in terms of creating a parser, is using recursive descent over a parser generator as it gives a more simplistic yet deeper understanding of grammars and parsing in general.

## Section 7: Software development life cycle model

The software development life cycle model used in this project was the Test Driven Development or TDD for short, where tests are used to drive development. For this project, we were given a set of tests that we needed to get passing to finish the compiler. This TDD style of development was very helpful in this project as it gave us goals to work towards, direction if we got stuck on something, and ultimately got us to the goal of finishing the project. Overall this TDD style of development was a great tool for learning more from this project and seeing what each unit of the compiler phases required.