

**Montana State University**

# **Capstone Portfolio**

**James Marsh, Jaclyn Saunders**

**CSCI 476 - Compilers**

**Professor Gross**

## Section One: Program

The source listing of the program I wrote for this course is attached as a zip file in the portfolio directory as source.zip.

### Grammar

Attached below is the grammar for Catscript. Catscript is a small, scripting programming language. It uses recursive descent all throughout the language. There are a few major components including types, expressions, statements, and more within this language.

```
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 , '>' ]

```

## Section Two: Teamwork

Although we did most of the project individually, there were some key teamwork aspects that added to the overall project and portfolio. Both team member one and team member two contributed the same amount of time on the teamwork sections of the project which amounted to about 10% of the time it took to complete the whole project. These sections included the technical writing document in section four, and three unit tests within the src zip file. We individually wrote our own tokenizers and parsers which allowed us to really understand the compiler process from start to finish. This took about 85% of the time it took to complete the whole project. The last 5% of time included writing this portfolio.

## Section Three: Design Pattern

```

// Memoization: more efficient by not creating a new list type every time, checks hashmap first
// is a type of caching, doesn't support multiple threads
2 usages
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
4 usages
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;
}

```

One design pattern that was used in my capstone project was memoization. This is located in CatscriptType.java in the getListType method. I used this method as an optimization technique in order to speed up the process of identifying Catscript types. The method works by storing each new type in a hashmap. When getListType is called, the hashmap is searched first in case the type is already pre existing. If so, it will just return that type out of the hashmap. If not, a new type is created and added to the hashmap.

## **Section Four: Technical Writing**

Attached PDF of technical writing piece from my partner...

# Catscript Guide

## Introduction

Catscript is a simple scripting language. Here is an example:

```
var x = "Hello world!"  
print(x)
```

This will output:

```
Hello world!
```

## Types

CatScript is statically typed. Variables can be create with the **var** keyword and assigned a specific type:

**int** - a 32 bit integer

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

**string** - a java-style string

```
var myString : string = "Hello world!"
```

**bool** - a boolean value

```
var x : bool = true  
var y : bool = false
```

**object** - any type of value

```
var x : object = 10  
var y : object = "Hello world!"  
var z : object = true
```

**list** - a list of values of the same type

```
var x : list<int> = [1,2,3]  
var y : list<string> = ["Dog", "Cat", "Bird"]  
var z : list<bool> = [true, false]  
var a : list<object> = [x, y, z]
```

**null** - the null type

```
var x = null
```

If a type is not specified, the type will be automatically assigned. In this example, **x** has type **int**:

```
var x = 10
```

## Features

### Print Statements

Catscript can print all objects to output:

```
var x : int = 10
var y : string = "Hello world!"
var z : bool = true
var a : list<int> = [1,2,3]
var b = null
```

```
print(x)
print(y)
print(z)
print(a)
print(b)
```

Output:

```
10
Hello world!
true
[1,2,3]
null
```

### Comments

Catscript allows users to include non-code comments that are ignored when the script is run. Everything to the right of the double slashes is ignored

```
var x = 10 // Example comment
print(x) // Comments do not affect the code
```

Output:

```
10
```

### For loops

Catscript can iterate through list variables:

```
var x : list = [1,2,3]
for (num in list) {
  print(num)
}
```

Output:

```
1
2
3
```

Catscript can also iterate through list literals:

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

Output:

```
1
2
3
```

## If Statements

Catscript can perform conditional operations using **if** statements:

```
var x = false
if (x == true) {
  print("fizz")
}
print("buzz")
```

Output:

```
buzz
```

Catscript can also branch using **else** statements:

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

Output:

```
buzz
buzz
```

## Comparisons

Catscript can check expressions for equality:

```
var x = 10
var y = 12
print(x == y)  // false
print(x != y)  // true
```

Catscript can also compare expressions:

```
var x = 10
var y = 12
var z = 10
var a = 12
print(x < y)    // true
print(x > y)    // false
print(x <= z)   // true
print(y <= a)   // true
```

## Functions

Catscript can be used to create reusable functions:

```
function hello() {
    print("This function is printing!")
}
print("Function Test:")
hello()
```

Output:

```
Function Test:
This function is printing!
```

Functions can return values of any CatScript type. The type is specified when the function is created, and returned with the **return** keyword:

```
function foo() : int {
    return 1
}
print("Function Test:")
print(foo())
```

Output:

```
Function Test:
1
```

Functions can also take in parameters. The parameter names are specified when the function is created, but the type is optional.

```
function add(y : int, z : int) : int {
    return y + z
}
print(foo(5, 6))
```

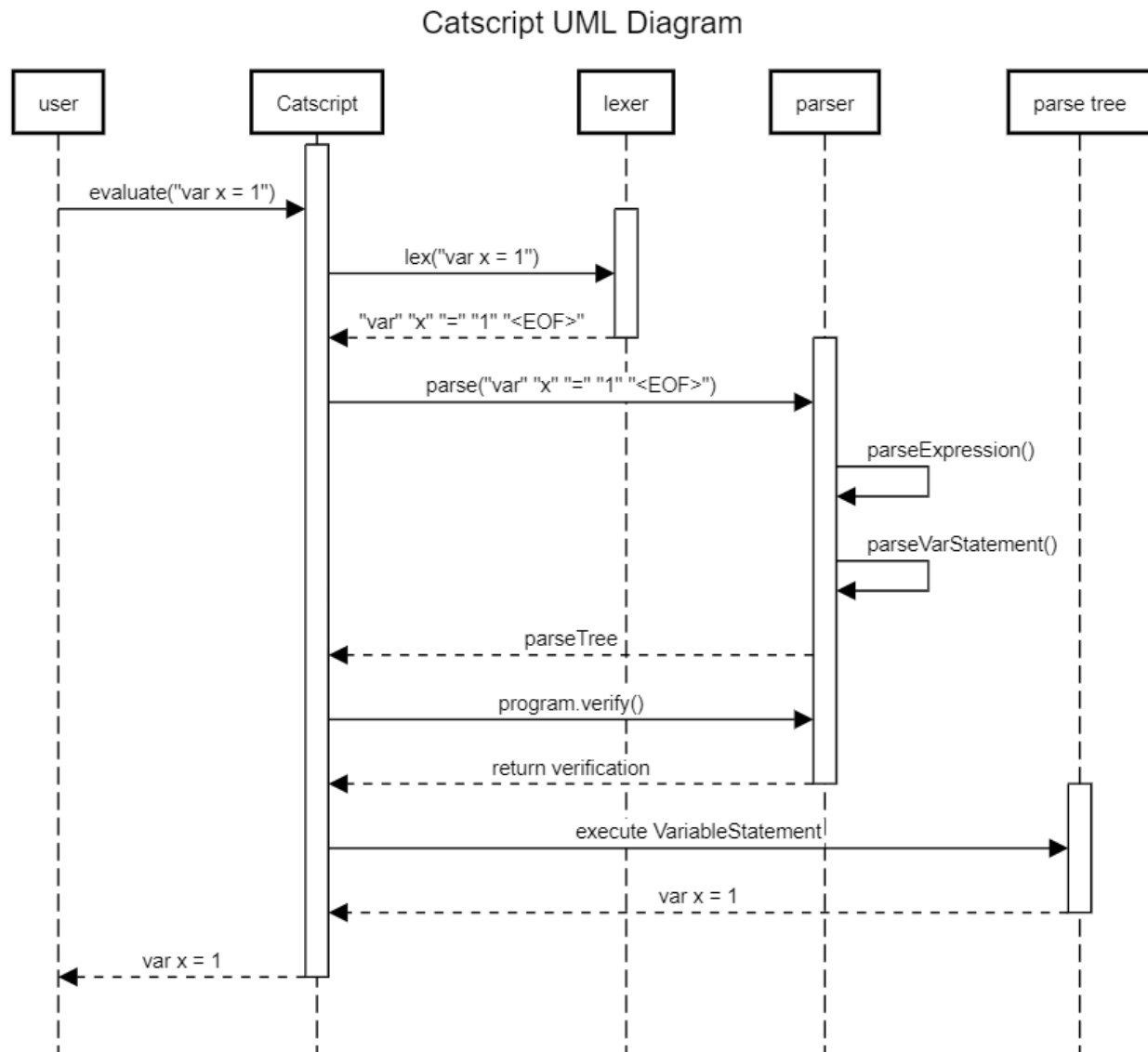
Output:

```
11
```



## Section Five: UML

This is a UML sequence diagram on how CatScript works. It first starts off with user input in the Catscript language, which then goes through a lexer to get back tokens which then parses and executes the expressions and/or statements and gives the result back to the user.



## Section Six: Design Trade-Offs

One of the design trade-offs I made was writing the entire parser by hand versus using a parser generator. I chose to do this in order to understand the parser better and use the recursive descent algorithm. Usually time efficiency is more important, but in this case, really understanding the code was more important to me than having a generator do it for me.

## **Section Seven: Software Development Life Cycle Model**

For this project, I used Test Driven Development (TDD) for the life cycle model. The tests were written first, and then the code was built around those tests. Ultimately, this allowed me to get the exact result and output that I wanted while building code around a certain goal. This type of life cycle model really helped me with the development of the software because it was an easy way to checkpoint my progress throughout the project. It really only hindered me when I couldn't get the output just right, but constant debugging and changing of the code allowed me to achieve my desired end result.