

# **MSU Computer Science Department Senior Portfolio**

**CSCI 468: Compilers  
Spring 2023**

**Alex Du Bois  
Eric Wanner-Garnier**

## Section 1: Program

See the source.zip file in this directory.

## Section 2: Teamwork

### Team Member 1 – Development

Team member 1 worked on development of the compiler and wrote the code implementation of its functionality.

### Team Member 2 – Documentation and Testing

Team member 2 created the documentation for CatScript grammar and features and provided tests for quality assurance.

## Section 3: Design pattern

The design pattern used in this project was memoization in the getListType function in parser/CatscriptType.java, line 37. This design was implemented to avoid continually creating new CatscriptType objects. Had we just coded it directly, every time we needed a list type, we would have created another object, and this would be inefficient use of memory. Using memoization, we can instead store already created list types in a Map and then access those rather than creating new ones. This means we will only create a finite number of objects (one for each type) thus saving us resources.

```
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;
}
```

## Section 4: Technical writing

### Introduction

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

```
var x = "foo"
print(x)
```

The Catscript parser was written using recursive decent, which should become apparent when reading the expression's syntax sections.

### Features

#### Catscript types:

int - a 32 bit integer

string - a java-style string

bool - a boolean value

list - a list of value with the type of it's contents

null - the null type

object - any type of value

### Statements

#### For loop:

General syntax:

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

Example:

```
var z = [1,2,3]
var counter = 0
for(x in z){
    print(x)
    counter = counter + 1
}
print(" Counter " + counter)
```

#### If statement:

General syntax:

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

Example:

```
if(1>2){
    print("one greater than 2")
}else{
    print("one is not greater than 2")
}
```

#### Print statement:

General syntax:

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

Example:

```
var x = 5
var z = 4
```

```
print(x+z)
```

### **Variable statement:**

General syntax:

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

Example:

```
var x = 5
var z:int = 4
print(x+z)
```

### **Assignment statement:**

General syntax:

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

Example:

```
var x = 2*5
var y = x
print(y)
```

### **Function declaration statement:**

General syntax:

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

Example:

```
function multiply(num1:int, num2:int){
    var answer = num1*num2
    return answer
}

print(multiply(2,5))
```

### **Function call statement:**

General syntax:

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

Example:

```
function multiply(num1:int, num2:int){
    var answer = num1*num2
    return answer
}

print(multiply(2,5))
```

## **Expressions**

### **Equality expression:**

General syntax:

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

Example:

```
print(1 == 2)
```

**Comparison expression:**

General syntax:

```
additive_expression { (">" | ">=" | "<" | "<=" )  
    additive_expression };
```

Example:

```
print(1 < 2)
```

**Additive expression:**

General syntax:

```
factor_expression { ("+" | "-" ) factor_expression };
```

Example:

```
print(2+3)
```

**Factor expression:**

General syntax:

```
unary_expression { ("/" | "*" ) unary_expression };
```

Example:

```
print(2*3)
```

**Unary expression:**

General syntax:

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

Example:

```
print(not true)
```

**Primary expression:**

This is the point in the language where we handle parenthesized expressions and this acts as both a reentry point for the grammar and the end point where our Catscript types are reached/attained.

General syntax:

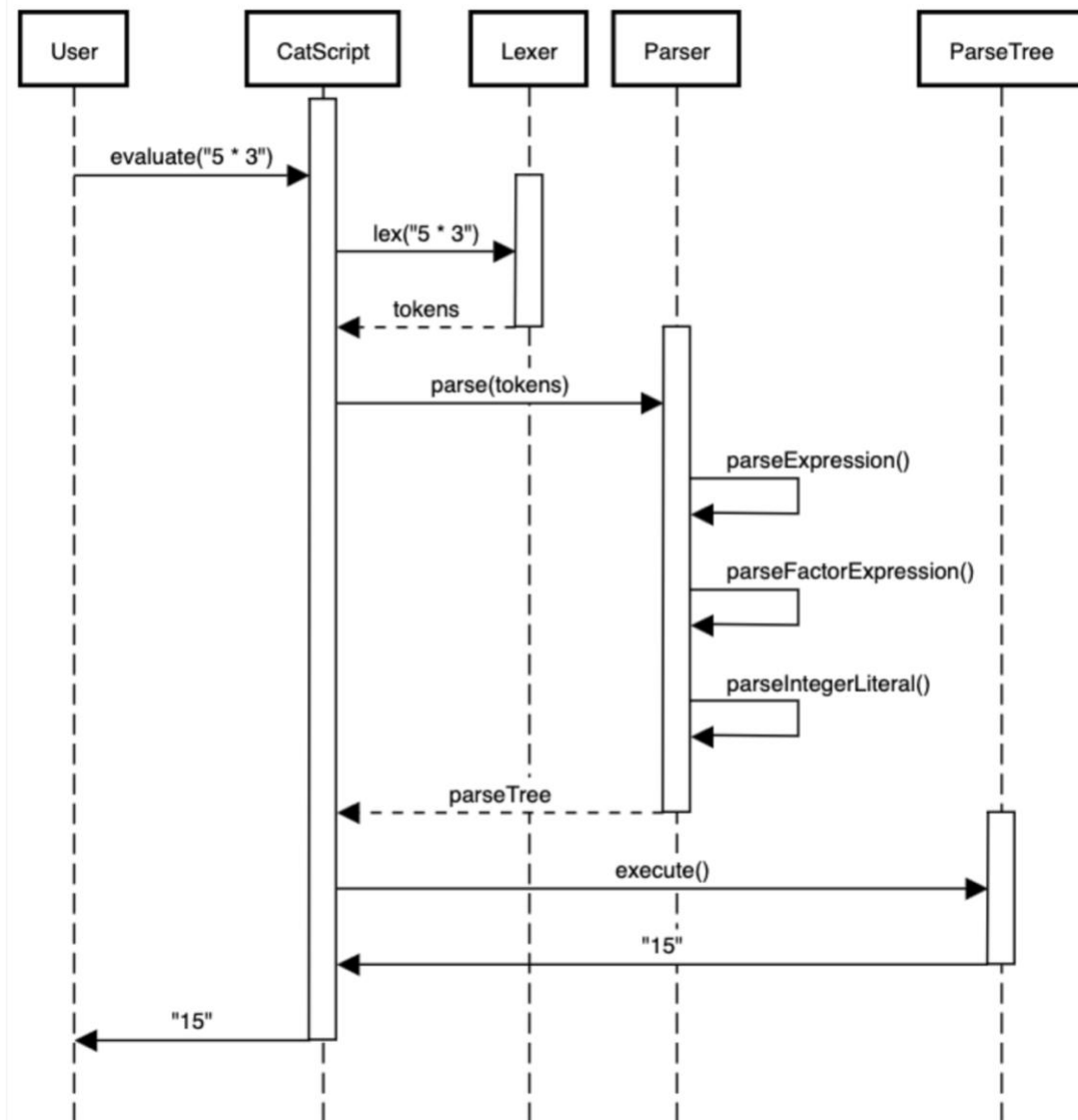
```
IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |  
list_literal | function_call | "(", expression, ")"
```

Example:

```
print(2*(5+2))
```

## Section 5: UML

Catscript Multiplication Sequence Diagram



This is a sequence diagram demonstrating how multiplication of 5 and 3 works in Catscript. We can particularly see the recursive descent nature of the Parser as it repeatedly calls parse functions while creating the parse tree.

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

One major trade off in this project was the fact that the parser was made using recursive descent, rather than generated through software. The reason we used recursive descent is that it is widely used in industry for developing programming languages, and clearly communicates the recursive nature of grammars.

The downside to using this method was that it involved much more intensive coding. Using a parser generator is more common in academic spaces, and would not require the same amount of programming. Instead, all that would be required is the grammar of the language. It would also be much faster to develop using a generator.

## **Section 7: Software development life cycle model**

The model used to develop our project was Test Driven Development. This meant that we had a suite of tests that needed to pass to ensure proper functionality of our parser. This model was very helpful – it provided direction of what to implement and when, and made it easy to tell whether we were on track. The downside to focusing so heavily on tests meant that in the beginning stages, it was easy to overlook issues that simply had no tests associated with them. These issues then needed to be resolved when the project moved to later stages in the compiler.