

Compilers Capstone 2023

Samuel Mocabee, Computer Science Professional

Montana State University Bozeman

Author Note

“Computers are incredibly fast, accurate, and stupid. Human beings are incredibly slow, inaccurate, and brilliant. Together they are powerful beyond imagination.”

- Albert Einstein, physicist

Abstract

The contents of this portfolio will go through a detailed report over the intricacies of the Catscript compiler. Technical documentation will be included to show readers a basic understanding of the Catscript language and function as a guide to learn how to code in Catscript. Teamwork was applied to the project as well, as team member one created additional tests to ensure the successfulness of executing programs in the Catscript language compiler. Several UML design diagrams are also included to show simplistic examples of how the compiler process works, and we will also look at a specific design pattern used in the process of coding and the trade-offs that came with those choices and their justification. Lastly will be viewing the software development life cycle model that was used to develop the capstone project.

Keywords: Source code, Teamwork, Design Pattern, Technical Writing, UML, Design Trade-offs, Software Development Life Cycle Model.

Section 1: Program

Attached to the document is a zip file of the final repository for CSCI 468, compilers. It holds the completed base source code for the compiler that was written over the course of the semester.

Section 2: Teamwork

Coding for this project was done individually allowing for team member 1 and I to get a deeper understanding of the compiler allowing us to work through the tokenization, parsing, and evaluation process at our own pace and through our own coding process. The primary contribution of team member 1 to this capstone project was through the tests that he wrote to ensure the validity of the compiler's structural ability and the technical documentation of the Catscript programming language.

Having team member one create additional test and the documentation for the language allows a separation in roles, as I function as the primary engineer responsible for the creation of the compiler and team member one who acts as the documentation and testing engineer. With this structure the roles are also reversed as I generate tests of my own to not only further evaluate my own compiler but to also be the role of the documentation and testing engineer and send the tests and technical documentation for team member 1's capstone project.

The following are the tests for my compiler created by team member 1. The first test makes sure that the if else statements work in a function declaration, the second test ensures that for loop and return statements work within a function declaration, and the third test checks complex expression evaluation and string concatenation.

```
package edu.montana.csci.csci468;

import edu.montana.csci.csci468.CatscriptTestBase;
import org.junit.jupiter.api.Test;

import java.util.Arrays;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNull;

public class CapstoneTests extends CatscriptTestBase {

    @Test
    void functionDefinitionWithIfStatementTest() {
        String input = "function foo(x : int) : int {\n" +
            "    if(x > 10){ print(2) }" +
            "    else{ print(x) }" +
            "    return x * 10" +
            "}\n" +
            "print(foo(9))";

        String expectedOutput = "9\n90\n";
        assertEquals(expectedOutput, executeProgram(input));
    }

    @Test
    void functionDefinitionWithForLoopTest() {
        String input = "function foo(x : int) : int {\n" +
            "    for(i in [1, 2, 3]){ print(i) }" +
            "    return x" +
            "}\n" +
            "print(foo(9))";

        String expectedOutput = "1\n2\n3\n9\n";
        assertEquals(expectedOutput, executeProgram(input));
    }

    @Test
    void complexExpressionEvaluationTest() {
        String input = "1 + 3 * 13 - (12 / 2)\n";
        String expectedOutput = "34\n";
        assertEquals(expectedOutput, executeProgram(input));

        input = "\"this\" + \"is\" + \"a\" + \"string\"\n";
        expectedOutput = "thisisastring\n";
        assertEquals(expectedOutput, executeProgram(input));
    }
}
```

Section 3: Design Pattern

One specific design pattern used in the capstone project is a memoization pattern located in the `CatscriptType.java` class in the function `getListType`. Memoization is an optimization technique to speed up computer programs by storing the results of function calls and returning the cached result when the same input occurs again. In the context of the compiler, when we call the `getListType` function with a `CatscriptType`; for this example let us use an `int` type. It investigates the hash map to see if the map contains the type of `int`. If not, it returns null and from there, we create a new list type or a list of `int` in this case and then puts this data into the hash map and returns the list type. Now if we run the function again the program will not re-initialize a list of type `int`, it will instead return the original list type we created. Allowing us to save computation of renewing a list type when it we call the function with the same type repeatedly. The following is the Memoization pattern code used in the project.

```
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

Catscript Programming Language Technical Documentation Provided by Team member 1

Introduction

Catscript is a small user-friendly programming language designed with simplicity in mind. It provides a straightforward syntax that enables programmers to write clear code. Catscript is a small functional and statically typed language. This documentation serves as a comprehensive guide to Catscript grammar and syntax. In this documentation, you will find detailed explanations of the various components of the Catscript language, including program structure, statements, control structures, functions, expressions, literals, and types. We will also provide examples and use cases to help you better understand the concepts and apply them yourself.

Recursive descent parsing is a top-down parsing technique used by Catscript to analyze and process its source code. This method starts at the highest level of the grammar and recursively applies the rules of the grammar to break down the source code into smaller components. The parser examines the input one token at a time, matching the grammar rules and constructing a parse tree in the process. If a rule cannot be successfully applied, the parser backtracks to a higher level in the grammar and attempts alternative rules. Recursive descent parsing is a popular choice for its simplicity and ease of implementation, which makes it well-suited for Catscript's limited features.

Types

In Catscript, types define the structure and behavior of data, determining the kind of values that variables can hold and the operations that can be performed on those values. The language offers a set of built-in types that cater to a variety of use cases, enabling you to create expressive programs. The language consists of seven distinct data types, namely: String, Integer, Boolean, List Literal, Object, Null, and Void. CatScript is statically typed and can be declared in variable statements and function declarations. All the types in CatScript are built on those defined in Java. At the time of evaluation CatScript also provides type checking in the validation step. When declaring variables or function parameters, you can optionally specify their type to enable type checking, which helps catch potential errors early in the development process. Types can also be used in generic contexts, such as defining lists of specific element types.

Integer:

In CatScript integers are built off the Integer.class in Java. It is important to note that in CatScript integers are the only numeric type that are provided. Any number that is provided is interpreted as a 32-bit integer.

Here is an example of declaring a variable as an integer and using it in an additive expression.

```
var x : int = 2  
  
x + 2
```

Strings:

In CatScript strings are built on the String class in Java but have some characteristics of their own. All strings are denoted with double quotes.

Here is an example of a variable string.

```
var str : string = "Hello World!"  
  
str + " I am a string!"
```

Boolean:

In CatScript strings are built on the Boolean class in Java. A Boolean literal in CatScript is put using the keywords “true” and “false”

```
var is_true : bool = true  
  
is_true == false
```

Lists:

In CatScript lists are built into the language as list literals and can be composed of any type. It is important to note that lists in CatScript can be composed of object types and therefore any mix of types.

Here are some examples of lists.

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

Objects:

In CatScript objects are built on the object class in Java. They are the root of all types and so can be the base for any type.

Null:

In CatScript the null type is also based on the object class in Java. Anything can be set to null in CatScript.

Expressions

In Catscript, expressions are fundamental building blocks used to represent values, perform computations, and evaluate conditions. Expressions can consist of literals (e.g., integers, strings, Booleans), variables, function calls, or combinations of these elements using various operators. The language supports a rich set of operators, allowing you to create complex expressions involving arithmetic, comparison, logic, and other operations.

Catscript expressions are organized hierarchically based on their precedence, which determines the order in which operations are performed. The precedence levels, from highest to lowest, include unary expressions (e.g., negation, logical NOT), factor expressions (e.g., multiplication, division), additive expressions (e.g., addition, subtraction), comparison expressions (e.g., greater than, less than), and equality expressions (e.g., equal, not equal).

When writing expressions in Catscript, parentheses can be used to group sub-expressions and explicitly define the order of evaluation, overriding the default precedence rules. Expressions are typically used as operands in various statements, such as assignments, conditional statements, loops, and function calls, allowing you to create powerful and expressive Catscript programs.

Logic Operators:

Within the CatScript language there are the usual logic operators available to the user, greater than, less than, greater than or equal to, less than or equal to, and equals.

Here are some examples of expressions using logic operators

```
x > 1
x < 4
x <= 5
x >= 2
x == 2
x == "foo"
```


Arithmetic Operators:

Catscript offers the basic operations of addition, subtraction, multiplication, and division. It is important to note that the addition operator can be used on integers or for string concatenation.

Here are a few examples.

```
1 + 1  
3 - 1  
4 / 2  
3 * 3
```

Statements

In Catscript, statements are the primary building blocks that define the behavior and flow of a program. They represent instructions that the program should execute, and are used to declare variables, define functions, control the flow of execution, and perform computations or other operations. Statements in Catscript are executed sequentially, from top to bottom, in the order they appear in the source code. However, control structures can alter the order of execution by branching or looping based on specified conditions. To group multiple statements together, you can use curly braces `{}` to create a block of code, which is particularly useful when defining the body of a function or the scope of a control structure.

For Loops:

In Catscript, for loops are used to iterate over a given range or collection, executing a block of code for each element. The loop variable is specified by an identifier and can be used within the loop body. For loops provide a convenient and readable way to perform repetitive tasks or process elements in a sequence.

Here is an example of how to write a loop in CatScript.

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

If Statements:

If statements are conditional constructs that allow you to execute a block of code if a specified expression evaluates to true. Optionally, you can include an `else` block to execute code when the condition is false. If statements are essential for controlling the flow of your program based on conditions.

Here is an example of how to use it for statements in CatScript.

```
if (x > 10)
{
    print(x)
}

else
{
    print(10)
}
```

Assignment Statement:

Assignment statements are used to assign a new value to an existing variable, identified by its name. This is a fundamental operation in any programming language, allowing you to store and update values in your program.

Here is an example of how to write an assignment statement after a variable has been created.

```
var x : int = 10

x = 5
```

Variable Statement:

Variable statements are used to declare new variables with a specified identifier and an initial value. Optionally, you can also define the type of variable, which provides type checking and helps to catch potential errors early in the development process.

Here is an example of how to instantiate a variable in CatScript.

```
var x = 10
var x : int = 10
var x : bool = true
var x : string = "hello"
var x : object = "foo"
```

Print Statement:

The print statement outputs the value of an expression to the console, allowing you to display information or debug your program. It is a useful tool for understanding the behavior of your code during development and testing.

Here is an example of the print statement.

```
print(10)
```

Function Definition Statements:

Function definition statements are used to create new functions with a specified identifier, parameter list, and a return statement. The function body consists of a series of statements that define the behavior of the function. Functions are essential for organizing your code into reusable, modular units.

Here is an example of defining a function in CatScript.

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

Function Call Statement:

Function call statements are used to invoke a previously defined function with a list of arguments. The function call statement evaluates the return value of the called function, which can be used in other expressions or statements.

Here is an example of how to call the function above.

```
print(foo(10))
```

Return Statements:

Return statements are used within function bodies to specify the value that should be returned by the function when it is called. A return statement can include an expression that evaluates the return value. If no expression is provided, the function returns `null`.

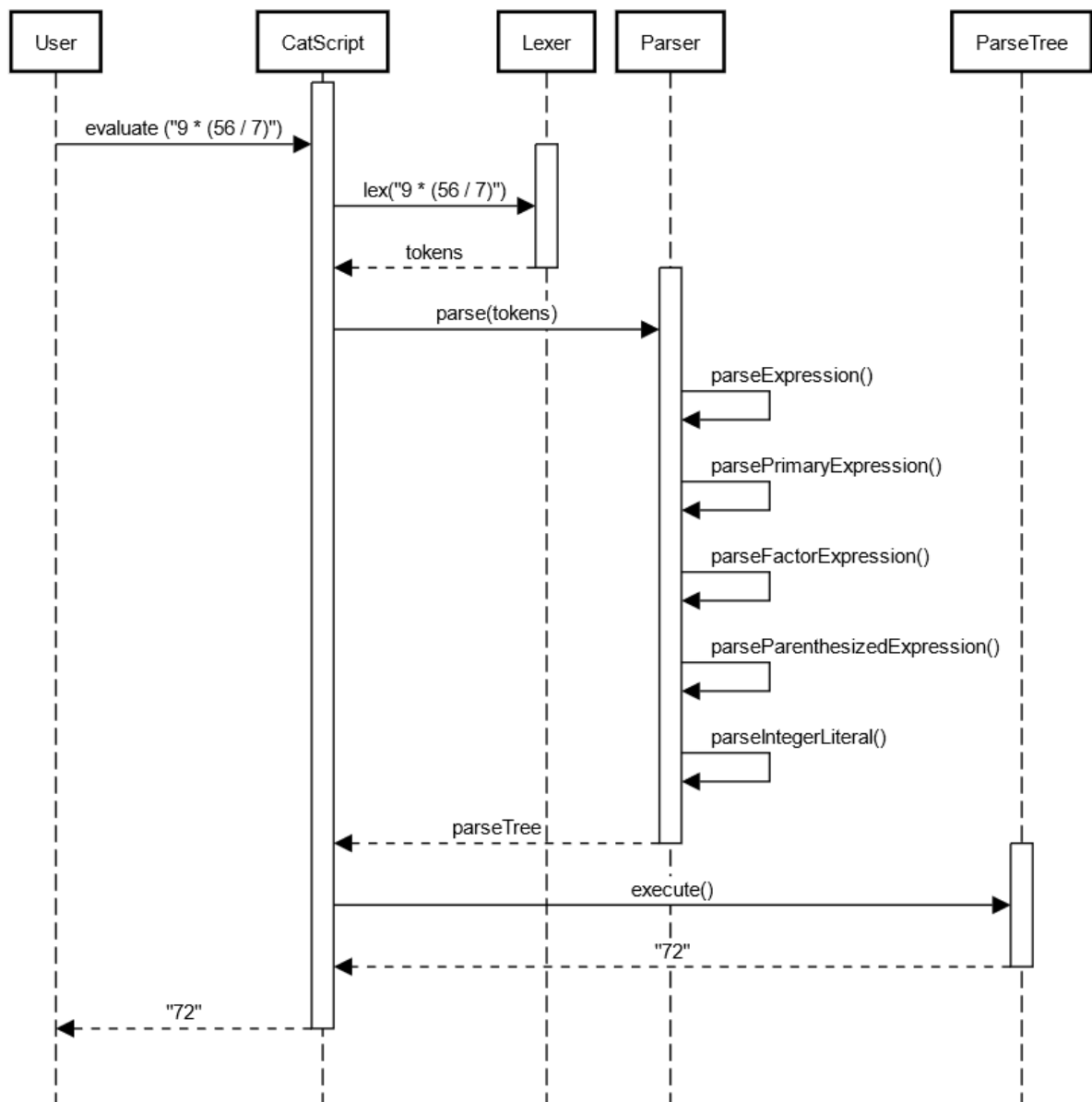
Here is an example of a return statement in a function.

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

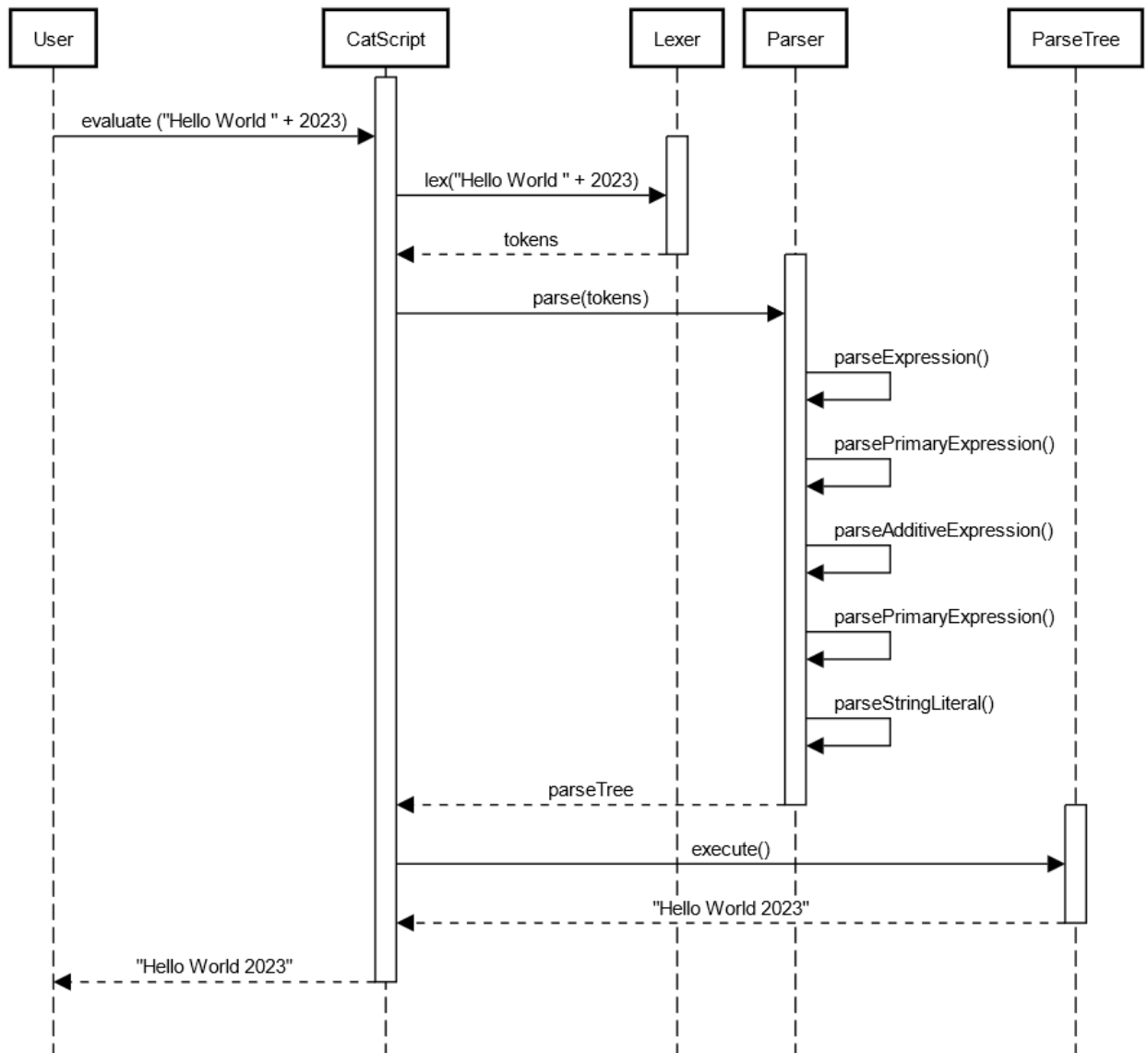
Section 5: UML

In this section I have included four sequence diagrams showing the process of parsing a numerical expression, a string concatenation expression, an if statement with a print, and a function definition containing a for loop with a print, a return statement, and a print with a function call statement.

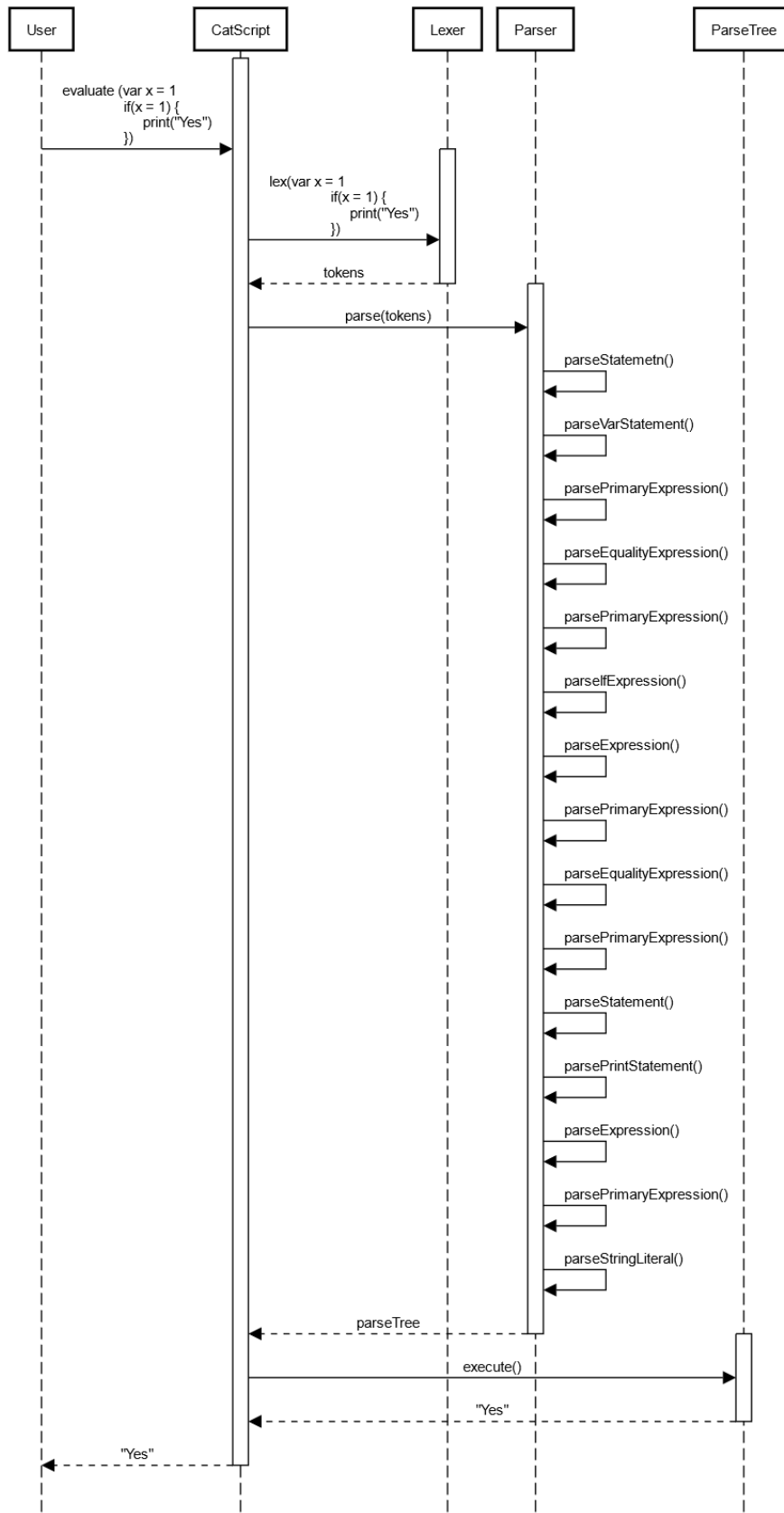
Catscript Parse Expression Sequence Diagram



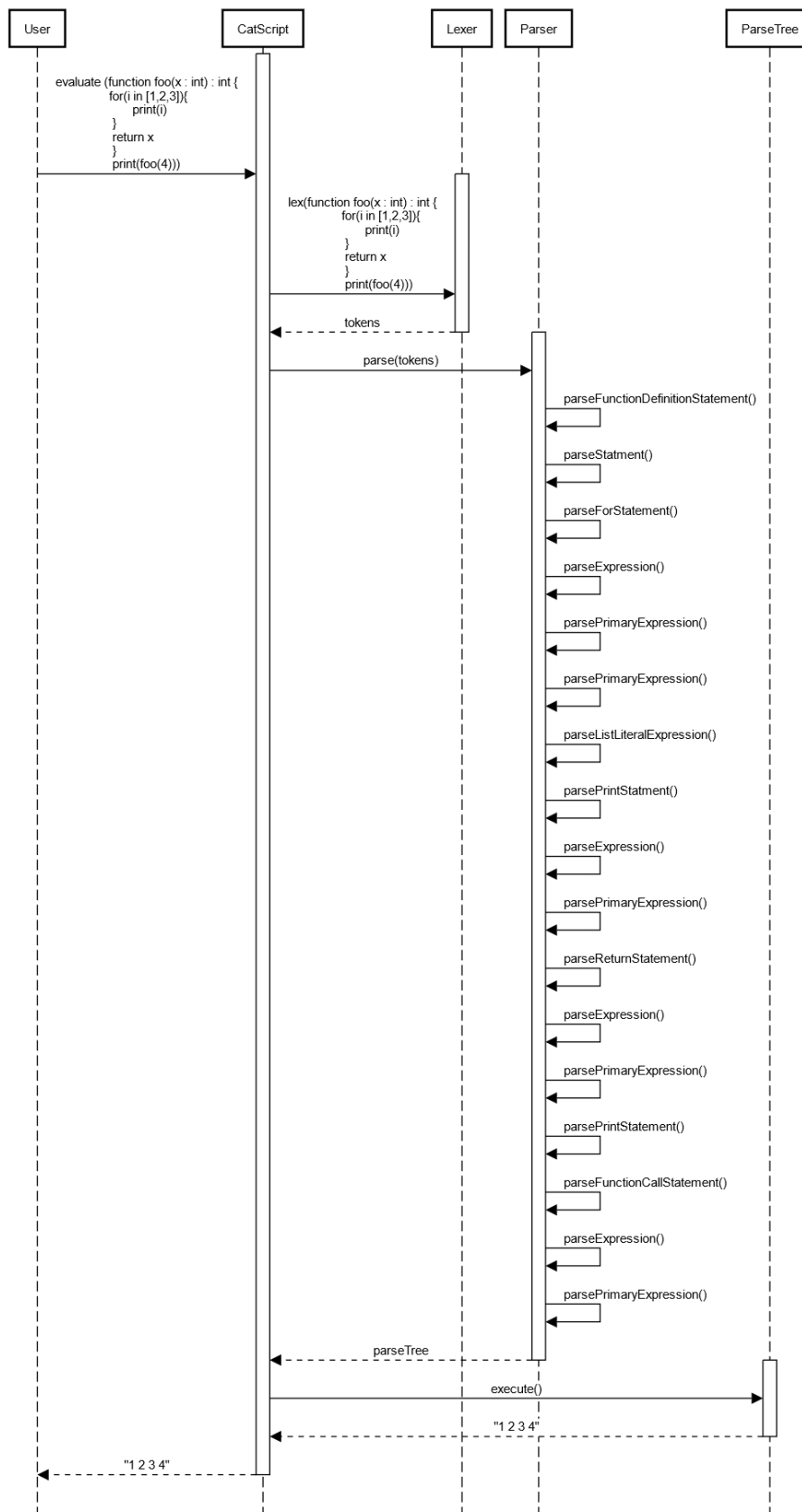
Catscript Parse String Expression Sequence Diagram



Catscript Parse If Statement Sequence Diagram



Catscript Parse Function Definition, For Loop, Return and Print Statment Sequence Diagram



Section 6: Design Trade-offs

The most notable design trade-off present in this capstone project was the recursive descent parser that was handwritten rather than writing lexical and language grammars to use a parser generator tool to create the parser. The justification of writing our parser for this capstone by hand was that we found this process more intuitive to observe a more in-depth view of how grammar works in compilers. Using a hand-written recursive-descent parser also allows coders in general to have complete control over the parser, enabling more opportunities that parser generators cannot accommodate, such as in-depth error messages and error recovery. It also gave us the chance to write more complicated code and taught us the process of which a compiler goes through to parse a language.

The comparison of the recursive decent method to the use of a variety of different parser generators tools like LAX, YAK, or ANTLR had a more educational advantage than using the technique of a parser generator program. Which would typically take the input in the form of a lexical grammar specified by a regular expression, and a language grammar in a EBNF (Extended Backus-Naur Form) syntax. While objectively leading us to a compiler that would parse and run Catscript just like the compiler created for the capstone, it robs us of looking at the bare bones of the parser process but also the chance to learn and create our own language for future coding endeavors.

Section 7: Software Development Life Cycle Model

The software development model that was used during the semester for the capstone project was test driven development which was an extremely helpful and useful model for this coding project. This software model continually tested previous test sections as well as the current section on which I was working. Each section of the tests would only pass if the previous sections were running properly, but as you progress you can have tests that uncover hidden issues that were missed but still passed in previous sections. For example, while working on the evaluation section one, a test was not executing properly because I had forgotten to adjust line-offset in a previous parsing function. Despite this error the tests in the parser section still passed. Using this development cycle allowed for step-by-step testing of the compilers capability while also checking previously coded sections.

When starting the process of creating our compiler we were given a series of tests that described the Catscript programming language. The tests for this project were divided into four sections: tokenizer, parser, evaluation, and bytecode. All these tests would eventually lead to a completed compiler, but to start this coding journey we focused on the tokenizer section. In this series of tests, a stream of text was broken into tokens while also attaching additional context to the tokens. For instance, the different operator types in the language or what the type of field of the token is, for example an int, bool, etc. The next section of development focused on parsing expressions and statements, while also type checking and symbol checking the tokens that were read into the parser, and asserting errors in the code if the given program did not follow the Catscript programming language. Once parse tests were evaluated correctly, we moved onto the evaluation process which would ensure the basic expressions, functions, and statements were generating the correct output based on the tests. Finally, the last sections of the tests would take

the Catscript code and would check to make sure that the basic expressions, functions, and statements were compiled correctly into bytecode. Ensuring the process of reading in the Catscript programming language, tokenizing the language, parsing, evaluating, and compiling the code down to byte code allowed us to successfully create our Catscript compiler for the capstone project.