

Compilers Capstone Portfolio

Emry Krems
CSCI 468
Professor: Carson Gross
Spring 2023

Section 1: Program

Specifications

The specifications for the program are listed below as the 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 , '>']

```

Program

Link to program source code:

https://drive.google.com/drive/folders/1OGbFXPe9fSUxW6vPZFqvV1V_kdThk0dy?usp=sharing

Section 2: Teamwork

There are two team members who contributed to this capstone project.

The primary engineer is Emry Krems. Emry wrote the code to produce the following components of the compiler:

- Tokenizer
- Parser for expressions and statements
- Evaluation for expressions
- Execution for statements
- Type checking
- Byte code

Emry also wrote sections 1-3 and 5-7 of the portfolio. The estimated percentage of time spent on this project by Emry is around 95%.

The testing and documentation engineer is Peyton Dorsh. Peyton provided a suite of tests to test some of the high-level functionality of the compiler. Peyton also wrote the documentation for CatScript (section 4) which outlines how CatScript works down to the expression and statement level. The estimated percentage of time spent on this project by Peyton is around 5%.

Section 3: Design Pattern

Throughout the computer science degree at MSU, students learn about design patterns in numerous classes, most notably, Software Engineering. In the compilers capstone project, one design pattern that is put into use is the memoization design pattern. The memoization design pattern speeds up computer programs by storing the results of function calls so each time the function is called in the future, the result does not need to be reproduced. Instead, this result is cached to be retrieved with every successive function call.

An example of where this design pattern is used in the compiler program is in the parser, specifically in the `CatScriptType.java` class. From lines 36-45, the class creates a hashmap where the key and value are `CatscriptTypes`. This hashmap holds list types so when the `getListType()` function is called with a certain type passed in, a new list type does not need to be created for that type. Instead, the function gets the hashmap with list types and checks if a list with that type already exists. If it does, the function returns that type from the hashmap. If a list with the type passed in does not exist, a new list with that type is created, added to the hash map for future function calls, and the list type is returned. The code of this design pattern is included below as well as being implemented in the compiler.

```
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: CatScript Documentation

Type System

CatScript is a statically typed language that features seven basic types.

- int - an integer
- string - a java-style string
- bool - a boolean value (true or false)
- list - a list of values with a template type
- null - the null type
- object - any type

Expressions

Additive Expressions:

The operators + and - are used to add and subtract two numbers. The + operator can also be used to perform string concatenation.

```
var x = 1 + 1;           // 2
var y = 2 - 5;           // -3
var z = "Fizz" + "Buzz"; // "FizzBuzz"
```

Factor Expressions:

The * operator is used to multiply integers and the / operator is used to perform integer division.

```
var x = 2 * 5;           // 10
var y = 10 / 2;          // 5
```

Unary Expressions:

The - operator is used to negate an integer. The not keyword is used to negate a boolean.

```
var someBool = not true;           // false
```

Comparison And Equality Expressions:

There are four comparison operators:

- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to

The two equality operators are:

- == equal to
- != not equal

Comparison expressions are used to compare two integers on the left-hand and right-hand side of the expression. Equality expressions test if two values are the same or different. Comparison and equality expressions are often used in if statements to provide control flow.

```
var x = 20;

if (x < 10) {
    // ...
}
// == being using for string comparison
var areStringsEqual = "Foo" == "Foo";
```

List Literal Expressions:

Lists are CatScript's supported collection and are immutable. Square brackets, [], are used to create a list of values belonging to the same type. All elements belonging to the list must be placed inside brackets.

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

```
var listOfStrings : list<string> = ["Fizz", "Buzz"];
```

Statements

Print Statement:

The most basic statement in CatScript is the print statement. It takes any expression as an argument and displays it to the console with a newline character appended to the end.

```
print("Hello, World!");    // Output: "Hello, World!\n"
```

```
print(10 + 20);           // Output: "30\n"
```

Unlike in most other languages, CatScript print statements are directly built into the grammar.

Variable Statements:

Variables store information in memory so information can be accessed after a value is computed. Variables can be of any type and are declared using the var keyword. The type of variable is automatically inferred by the compiler, but it can also be explicitly stated.

```
var x = "String";
```

```
var y: string = "String";
```

Variables in CatScript are *mutable*.

```
var x = 100;
```

```
var y = 50;
```

```
x=x +y;
```

```
print(x);    // Output: "150\n"
```

Variable names (identifiers) must start with a letter and must not contain special characters.

```
myVar => valid
```


Variable123 => valid

12var => not valid

num@ => not valid

If Statement:

CatScript uses if statements for all forms of control flow. They comprise an expression that evaluates to a boolean value and a body. If the expression evaluates to true, the body is executed.

```
var x = 5;

if (x > 2) {
    print("x is greater than 2!");
}
```

// output: "x is greater than 2!\n"

They can also be chained together through the use of the else keyword. A common control flow pattern is this:

```
var x = 125;

if (x < 100) {
    print("X is less than 100");
} else if (x >= 200) {
    print("X is greater than or equal to 200");
} else {
    print("X is in the range (100, 200]");
}
```

// output: "X is in the range (100, 200]\n"

An arbitrary number of else if s can be chained together. The ending else must be at the end of the flow.

For Statement:

For statements provide the ability to iterate over a list of elements. They comprise a header of an identifier for the current element and a list. This list can be sourced from a literal expression of a variable of type list. The identifier and list are joined by the *in* keyword.

The following two for statements are identical:

```
// For loop using a list literal
for (x in [1, 2, 3]) {
    print(x);
}
// For loop using a variable of type list
var someList = [1, 2, 3]
for (x in someList) {
    print(x);
}
```

They can also be composed together where a for statement comprises the body of another.

```
var exampleList = [20, 50, 90];
for (i in exampleList) {
    for (j in [5, 10]) {
        print(i + j);
    }
}
// Output: "25\n30\n55\n60\n95\n100\n"
```

Function Declaration Statement:

Functions are groups of code that perform a certain task. They are declared with the function keyword and can take zero or any number of specified arguments. These arguments can have their type inferred or explicitly declared. Functions can optionally return a value by adding a colon followed by the type after the arguments are listed.

Inside the brackets is the body of the function where any non-function declaration statement can be written.

// function that returns and takes no arguments

```
function foo() {
    print("FOO");
}
```

// function that returns the string "BAR"

```
function bar() : string {
    return "BAR";
}
```

// function that takes in three arguments and returns an integer

```
function foobar(x, y: int, z: list<int>) : int {
    // ...
}
```

Function Call Statement:

Calling a function means executing its body. To call a function use its identifier followed by parenthesis. Inside these parentheses, the function arguments are provided in order. You can optionally store or print the value returned by a function.

// no arguments or return type
foo();

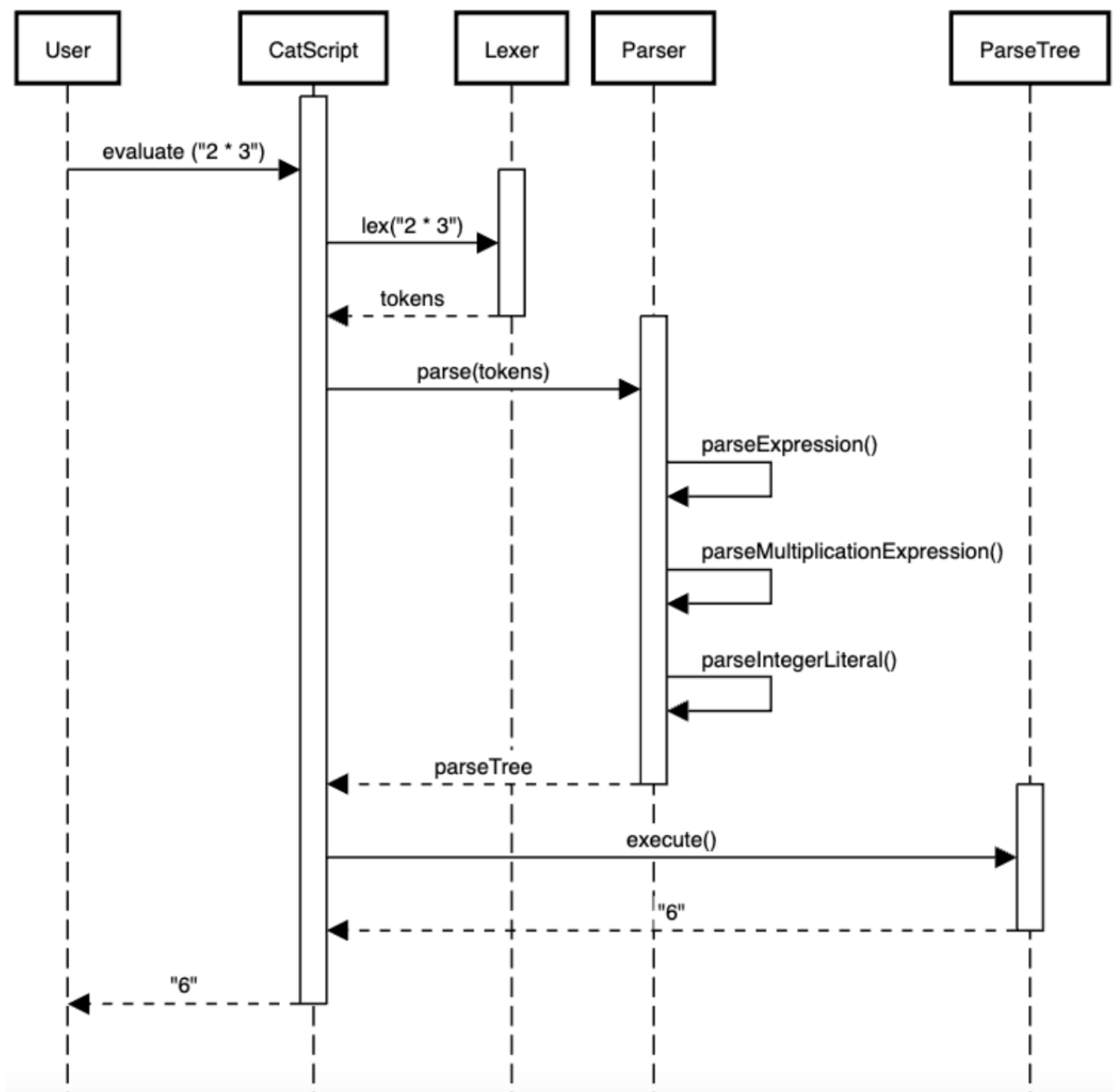
// no arguments and stores the return value in variable a
var a = bar();

// two integer arguments and one list argument and stores the output in variable b
var b = foobar(10, 20, [1, 2, 3]);

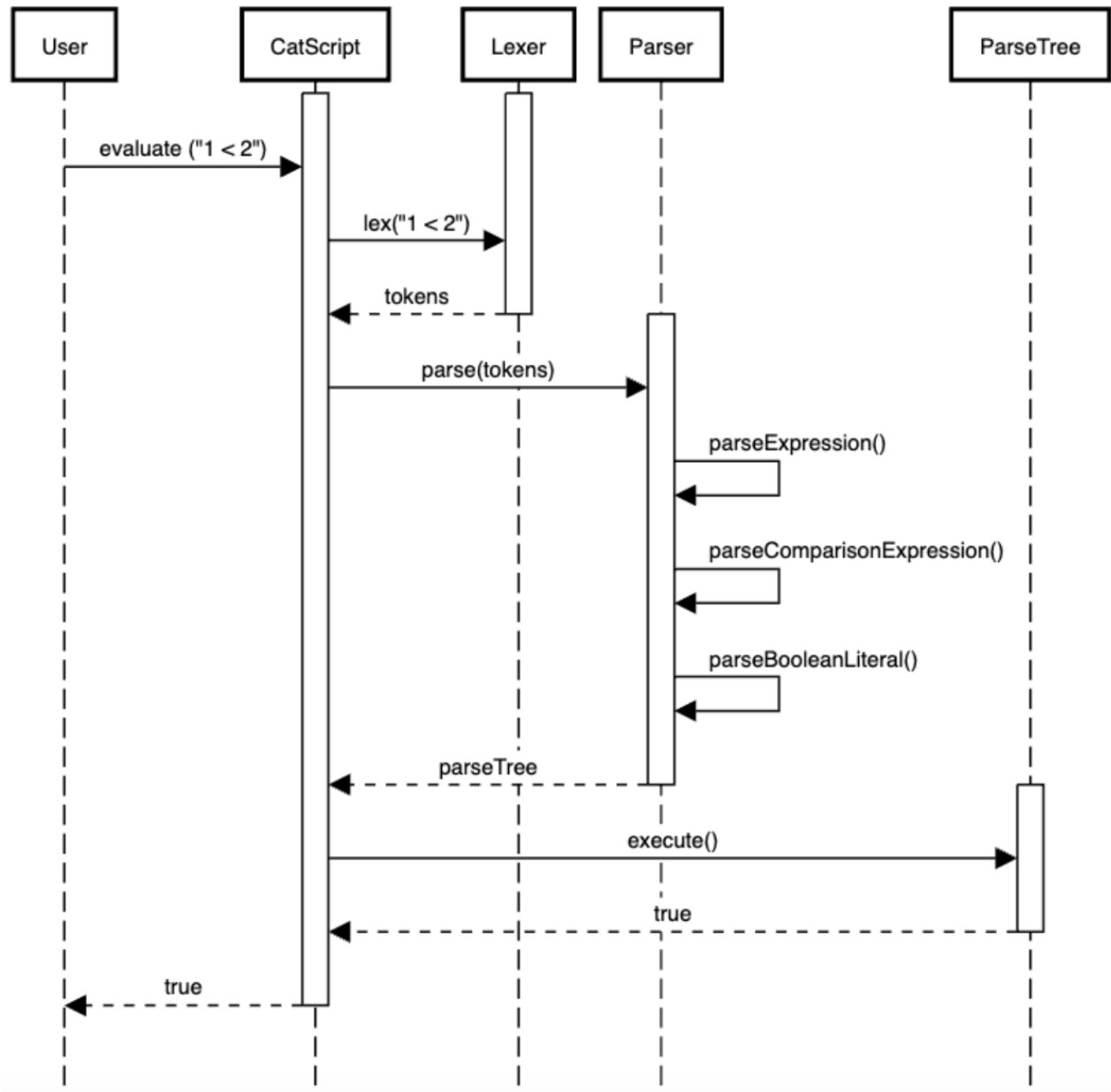
Section 5: UML

Similarly to the design patterns described in section 3, throughout the computer science degree at MSU, students learn about UML diagrams and their use for designing software prior to beginning to write code. The following 3 UML sequence diagrams show examples of what the design looked like for the multiplication expression, comparison expression, and if statement in the compiler.

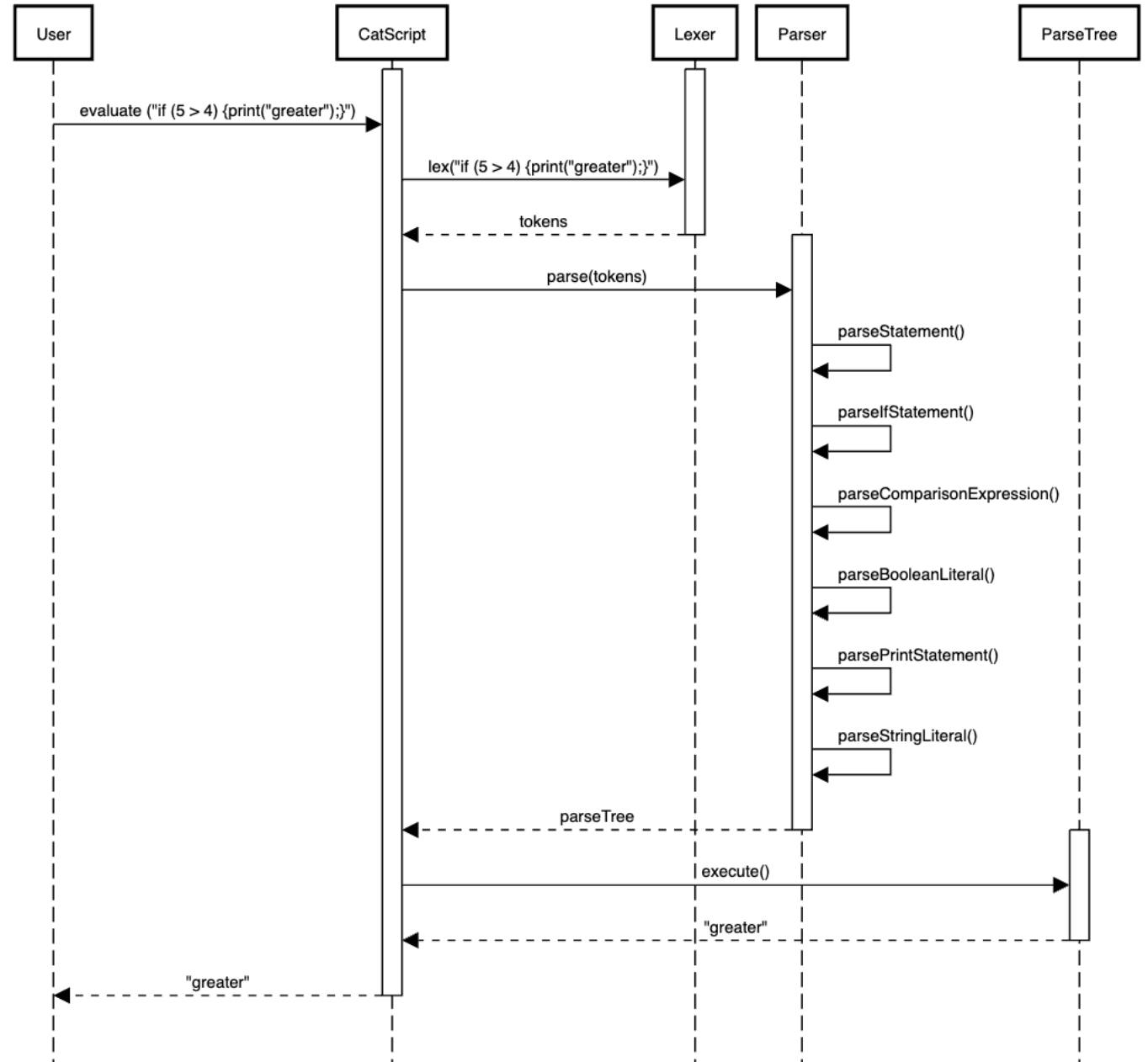
Catscript Multiplication Sequence Diagram



Catscript Comparison Sequence Diagram



Catscript If Statement Sequence Diagram



Section 6: Design Tradeoffs

The major design tradeoff when writing the compiler for this project was deciding to use recursive descent parsing over using a parser generator. Most compiler courses at other universities would use a parser generator.

A parser generator is a computer program that takes a language specification and produces a parser for that language. The input for this parser is broken up into a lexical grammar and a language grammar.

For the parsers in this compiler, a recursive descent parsing algorithm is implemented. A recursive descent parser is a top-down parser where each recursive function implements one of the non-terminals in the grammar. This kind of parser must be written by hand.

The reason a recursive descent parser was chosen for this project over a parser generator was that the students received a more well-rounded education on how to write a compiler. Writing a parser by hand forced the student to understand how to read and implement a grammar. Additionally, by writing the parser by hand, debugging becomes much easier. If a parser generator were used to create the parser for this compiler, the student would not be familiar with the inner workings of the compiler which would make it difficult to debug as effectively. Parser generators also write parsers that are much more convoluted than a recursive descent parser.

For simplicity and an elevated knowledge of how parsers work, a recursive descent parser is the preferred design.

Section 7: Software Development Lifecycle Model

The software development lifecycle used for this compiler is the Test Driven Development Model (TDD).

The TDD is a model that turns software requirements into test cases before the code is written to meet the requirements. TDD benefits the developer because the tests serve as a guide to building up functionality in the simplest way possible. This model may slow down development in the beginning phases, but in the long run, development becomes much faster as the functionality of the code can be immediately tested for accuracy.

For this capstone project, there was a test suite provided for each of the different building blocks of the compiler (i.e. tokenizer, parser, etc.). Since the tests were provided at the beginning of the project, the primary engineer did not experience a slow start to development by having to write the tests. Instead, coding for functionality in accordance with the desired output from the tests could begin immediately. This was extremely beneficial for the developer as they knew whether or not the code was performing as expected directly after writing. This eliminated the need to spend time trying to test functionality in more time-consuming ways. The tests also aided in debugging the code. When a test would fail, breakpoints could be added to the code to watch the execution in real-time to see where the code was not behaving as desired.

The use of the TDD model for the compilers capstone introduces students to an invaluable development lifecycle model as TDD is extremely effective in prompting the developer to write concise code with the correct functionality as specified by design requirements.

Below is an example of the tests used to drive development. These tests were written by Peyton Dorsh, the testing engineer.

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

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

import static org.junit.jupiter.api.Assertions.*;

public class CapstoneTests extends CatscriptTestBase {
    @Test
    void nestedForLoopsTest() {
```



```

        assertEquals("15\n", executeProgram(
            "var x = 0\n" +
            "for (i in [1, 2, 3, 4, 5]) {\n" +
            "for (j in [1, 2, 3]) {\n" +
            "x = x + 1\n" +
            "}\n" +
            "}\n" +
            "print(x)"));
    }

@Test
void passingArgumentsInsideFunctionsTest() {
    assertEquals("FooBar\n", executeProgram(
        "function foo(x) {\n" +
        "bar(x)\n" +
        "}\n" +
        "function bar(x) {\n" +
        "print(x)\n" +
        "}\n" +
        "foo(\"FooBar\")"));
}

@Test
void mutableVariableInsideLoopTest() {
    assertEquals("4\n100\n", executeProgram(
        "var x = [5, 4, -2, 4, 2, 100, 10]" +
        "var prev = 100\n" +
        "for (i in x) {\n" +
        "if (prev < i) {\n" +
        "print(i)\n" +
        "}\n" +
        "prev = i\n" +
        "}\n"));
}
}

```