

Section 1: Program

A zip file of the source code has been included

Section 2: Teamwork

For this Compiler project, Team member 1 wrote the tokenizer, parser and compiler in Java. Team member 2 wrote the documentation and and three tests to ensure the parser works correctly. The three passing tests are below:

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

import edu.montana.csci.csci468.CatscriptTestBase;
import org.junit.jupiter.api.Test;
import java.util.Arrays;
import static org.junit.jupiter.api.Assertions.*;

public class partnerTests extends CatscriptTestBase {
    @Test
    void nestedLoopsWork() {
        assertEquals("1\n2\n3\n1\n2\n3\n1\n2\n3\n", executeProgram("for( x in [1, 2, 3] ) {\n" +
            "    for( y in [1, 2, 3] ) {\n" +
            "        print(y)\n" +
            "    }\n" +
            "}\n"));
    }

    @Test
    void ifStatementWorksInForLoop() {
        assertEquals("3\n", executeProgram("for( x in [1,2,3]) {\n if(x==3){\n print(x)\n}\n}"));
    }

    @Test
    void forLoopWorksInIfStatement() {
        assertEquals("1\n2\n3\n", executeProgram("var x = 10\n if(x>5){\n for(y in [1,2,3]){ \n print(y)\n}\n}"));
    }
}
```

Section 3: Design Pattern

In the Catscript parser, there's a function to return the type of a list. Within the function, memoization is used to speed up subsequent calls to the function during the validation stage. We used the memoization pattern as this function can be slow and memory intensive with generic types. The code below can be found in CatscriptType.java on line 39

```
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) {
        return listType;
    } else {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
        return listType;
    }
}
```

Section 4: Technical Writing

Below is the technical document that accomanied my capstone project

Introduction

Catscript is a simple, statically typed scripting language. Here is an example:

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

Datatypes

There are five main datatypes in Catscript, those being integers, strings, booleans, list, and objects. There are also null values in Catscript, which can be stored as any datatype. Datatypes in Catscript can be defined either explicitly as follows:

```
var x: int = 5
```

or implicitly as follows:

```
var x = 5
```

Integers

Integers in Catscript are whole number values which are stored as 32 bits and can be defined in variables using the "int" keyword as follows:

```
var x: int = 5
```

Strings

Strings in Catscript are similar to strings in java - any string of characters enclosed by double quotation marks. They can be defined using the "string" keyword or implicitly:

```
var x: string = "hello world!"
```

Booleans

Boolean values in Catscript can have either the value "true", "false", or "null". They can be defined using the "bool" keyword:

```
var x: boolean = true
```

Lists

Lists in Catscripts are variable length collections of values that can be defined by square brackets enclosing comma separated values as follows:

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

As shown above, the list type is explicitly defined using the "list" keyword followed by angle brackets enclosing the component type of the list.

Lists may only include one component type.

Objects

Objects in Catscript can store any of the aforementioned datatypes and are defined using the "object" keyword:

```
var x: object = null
```

Variables

Variables in Catscript are initialized and defined as seen above, optionally followed by an explicit type expression.

Variable names must start with a letter, but may contain either letters or numbers.

Underscores and other special characters are not permitted in Catscript variables.

Examples:

```
var x: int = 5
var y = "hello world"
```

The value a variable holds can be changed using an assignment statement:

```
var x = 5
x = 10
```

Comments

Comments in Catscript can be denoted using two slashes in a row and take up a single line.

Example:

```
//this is a comment
```

Expressions

Expressions in Catscript are made up of either variables or literals as well as operators, typically infix except for in the case of the two unary operators "not" and "!". One notable exception to either of these formats is the type expression.

There are eight types of expressions in Catscript.

Primary Expressions

A primary expression in Catscript consists of any of the aforementioned literal expressions for integers, booleans, or lists as well as identifiers and can also be enclosed in parentheses.

Here are some examples of primary expressions:

```
true
false
null
"hello world"
x
[1,2,3]
(true)
```

Unary Expressions

The two unary operators in Catscript are "-" and "not". The unary "-" operator will negate a number and the unary "not" operator will negate a boolean value.

Unary Expressions in Catscript consist of one of these unary operators followed by either a primary expression or another unary expression, like so:

```
-10
-(-10)
not true
```

Factor Expressions

In Catscript, factor expressions are used to multiply and divide two unary expressions.

Examples:

```
x*y
10/5
```

Additive Expressions

Additive expressions are similar to factor expressions, except that they are used to add or subtract two numbers.

Additive expressions can also be used to concatenate two strings together.

Examples:

```
x+y
"hello" + "world"
```

```
"hello" + null  
y-x
```

Comparison Expressions

Comparison expressions are used to compare two other expressions. There are four operators used in comparison expressions:

The less than operator: "<"

The greater than operator: ">"

The less than or equal to operator: "<="

and the greater than or equal to operator: ">="

These operators determine if the lefthand expression is less than, greater than, less than or equal to, or greater than or equal to respectively.

Comparison expressions evaluate to boolean values.

Examples:

```
var x = 5  
var y = 10  
x < y  
x <= y  
x > y  
x >= y
```

Equality Expressions

Equality expressions in Catscript are similar to comparison expressions, but they only have two operators.

The equality operator: "==" and the not equals operator: "!="

If two expressions are equal, the equality operator will return true.

If two expressions are equal, the not equals operator will return false.

Example:

```
var x = 10  
var y = 10  
x == y //true  
x != y //false
```

Statements

Statements in Catscript are made up of expressions, keywords, and other symbols.

Statements produce output, mutate variables, or branch in different ways during the execution of a Catscript program.

There are six types of statements in Catscript. Since we have already discussed variables, we will talk about the

remaining four types of statements below.

We will also discuss function declarations and bodies as they each contain statements or expressions as well.

Print Statement

Print statements in Catscript use the "print" keyword followed by an expression enclosed in parentheses.

They will print the value of the expression to the output.

Example:

```
var x = 5
print(x)
```

If statement

If statements in Catscript use the "if" keyword followed by an expression enclosed in parentheses and a body of statements enclosed in curly braces.

Optionally, an block of "else" statements can also be defined using the "else" keyword and more curly braces.

If the expression in the if statement evaluates to `javascript true`, the first block of statements will be run.

If the expression evaluates to `javascript false` and there is an else block, the statements in the else block will be run before continuing to the next line.

Example:

```
if(true) {
  print("true")
} else {
  print("false")
}
```

For Loops

For loops in catscript are defined using the "for" keyword followed by parentheses enclosing an identifier to be used in the loop, the keyword "in" and either a list literal or an identifier that evaluates to a list. The first variable inside the parentheses will be assigned to the next value in the list at each step of the loop until there are no more items in the list.

Following the parentheses, there is a block of statements enclosed in curly braces that will be run during each iteration of the loop.

Example:

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

Functions and Function Declarations

Functions in Catscript can be declared using the "function keyword keyword followed by an identifier and a set of parentheses.

Optionally, a set of parameters can be declared inside of the parentheses to be used in the function. Function parameters can be given an optional explicit type during declaration, similar to variables.

After the parameters, a block of statements enclosed in curly braces denotes the function body, which is executed each time the function is called.

Functions can have the following return types:

int: an integer value

string: a string value

list<component type>: a list containing values of the type "component type"

object: any data type void: the function does not return anything.

Example:

```
function foo(x: int) {  
    print(x)  
}
```

Function Call Statement

A function call statement in Catscript consists of an identifier that corresponds to the name of a declared function followed by a set of parentheses that enclose any parameters being passed in to the function. When a function is called, all statements in its body will be executed in the order they appear.

Example:

```
foo(5)
```

Return Statement

Return statements in Catscript can only be used inside of a function body.

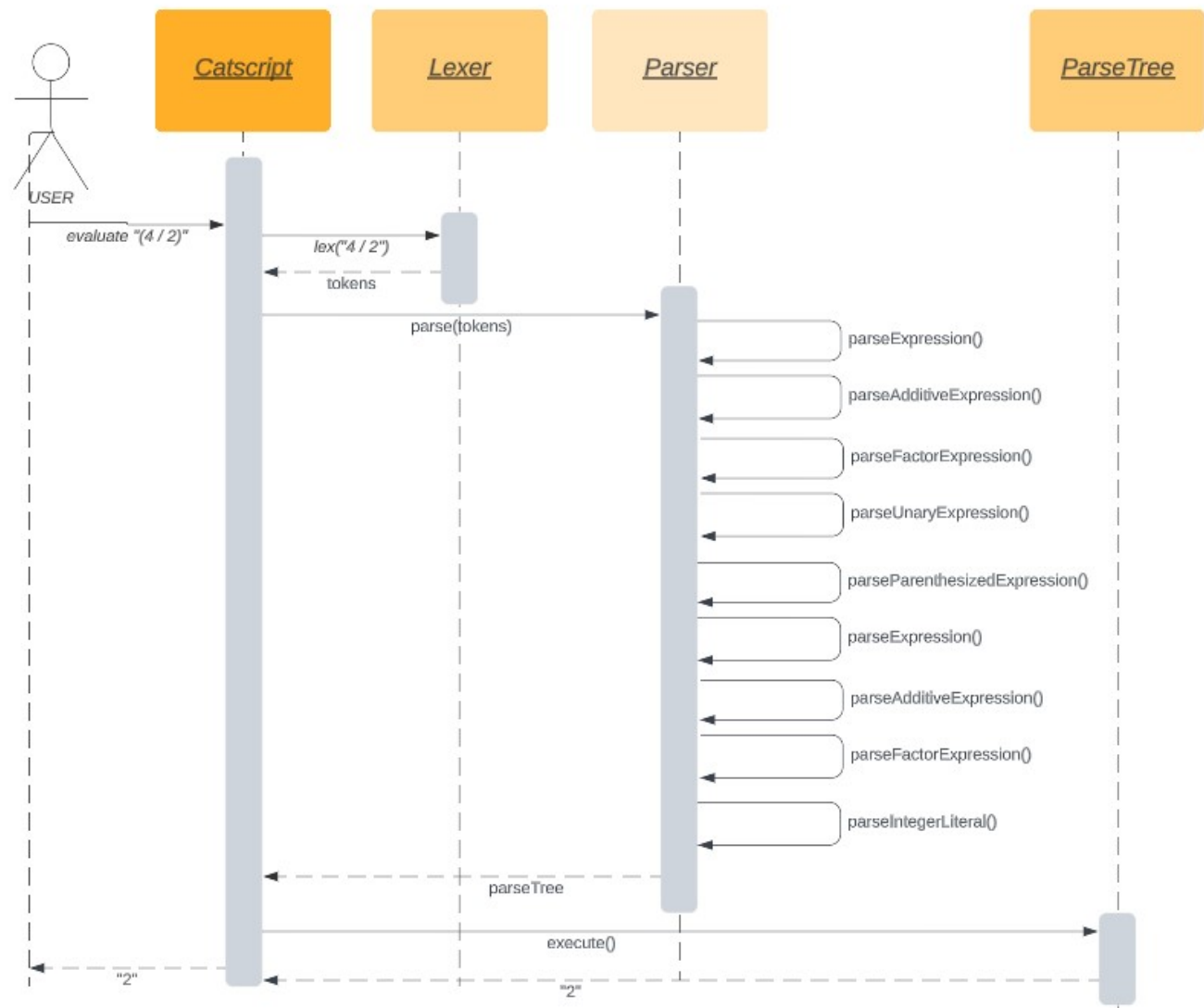
These statements are denoted by the "return" keyword followed by an expression and are used to return the value of the expression from the function to be used later in the program or to be assigned to a variable.

Example:

```
function foo: int(x: int) {  
    return x  
}  
var y = foo(5)
```


Section 5: UML

A sequence diagram of a parenthesized factor expression in Catscript is as follows:



Section 6: Design Trade-offs

In the CSCI 468 - Compilers class, we used the recursive descent algorithm to write a parser by hand. We did this manually as opposed to using a parser generator as to give us a better understanding of the function of parsers, as well as being easier to use the debugger to identify and fix bugs throughout.

Section 7: Software Development Life Cycle

Within the CSCI 468 - Compilers course, we used Test Driven Development as our life cycle model. Using test driven development, our team was able to work from the ground up in ensuring our tokenizer, parser and compiler had no bugs and worked in the desired fashion. The only negative impact realized by using test driven development was the time spent running tests in between writing and fixing code. Using unit tests

would alleviate some of this time, although the same problems exist for most testing strategies. The best part of the test driven development was the immediate feedback, knowing whether the parser, tokenizer or compiler were correct based on if any tests failed or not. Test Driven Development worked extremely well for our use case.
