Section 1: Program

Source code is attached.

Section 2: Teamwork

I wrote documentation for my partner's project and subsequently created new test cases for their code. This process allowed us to identify potential issues or areas for improvement, ultimately enhancing the functionality of our codebase.

Likewise, my partner reciprocated by providing a thorough documentation for my code and contributed additional test cases. This iterative process of collaboration and feedback not only facilitated the detection and resolution of potential defects but also fostered a deeper understanding of each other's work, leading to more effective communication and cooperation.

Section 3: Design pattern

An application of design patterns can be found in the CatscriptType.java file, which contains a method named getListType() that leverages the memoization pattern to enhance the efficiency and performance of the program.

Memoization is a technique that involves caching and reusing the results of expensive function calls, thereby reducing the overall computational overhead. In the context of our getListType method, we utilized a HashMap data structure to store previously computed listTypes. By storing the listTypes in a HashMap, we effectively eliminated the need to create new instances of listType during each method call, ultimately reducing both memory consumption and processing time.

Section 4: Catscript Guide

Introduction

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

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

Table of Contents

Features

Catscript implements the basic features one expects in a programming language. These features can be separated into two categories: statements and expressions. Statements perform actions or control the flow of the code, whereas expression evaluate to a single value. Function declaration statements are separated from the Statements section as they are more nuanced and are made up of multiple other components.

Statements

For Loop Statement

A for loop is declared using the keyword "for" followed by a set of parentheses containing an identifier, the keyword "in", and an expression. The expression can be anything that produces a list of values, such as a list literal or a function call that returns a list. The for loop then executes a sequence of statements contained within curly braces for each item in the list produced by the expression.

> Note that the identifier does not need to be declared beforehand.

The example below shows how to use a for loop to print the numbers 1-5 in Catscript.

```
for (i in [1, 2, 3, 4, 5]) {
  print(i)
}
```

If Statement

An if statement is declared using the keyword "if" followed by a set of parentheses containing a boolean expression. If the expression evaluates to true, then the statements contained within the curly braces following the if statement will be executed. Optionally, an else clause can be included that contains statements to be executed if the expression evaluates to false.

In this example, the if statement checks whether the value of the variable x is greater than 5. In this case, the string "x is greater than 5" will be printed.

```
var x = 10
if (x > 5) {
  print("x is greater than 5")
}
```

The following example shows how to use an if statement with an else clause. In this case, the string "You can vote!" will be printed.

```
var age = 19
if (age < 18) {
  print("You are not old enough to vote.")
} else {
  print("You can vote!")
}
```

Print Statement

A print statement is used to write a value to the output. It is declared using the keyword "print" followed by a set of parentheses containing an expression to be printed. The expression can be a variable, a literal value, or the result of an expression.

> Note that the string value of the expression will be written to output, so print statements are not limited to strings.

In this example, the print statement writes the string "Hello, world!" to output:

```
print("Hello, world!")

// Output: Hello, world!
```

In this example, the print statement displays the result of a calculation:

```
print(2 + 3)

// Output: 5
```

Variable Statement

A variable statement is used to declare a variable and assign it an initial value. It is declared using the keyword "var" followed by an identifier, an optional type annotation, an equal sign, and an expression. The expression can be a literal value, the result of an expression, or a function call that returns a value.

> Note that the type annotation is optional, but it can be used to specify the type of the variable.

In this example, a variable called "y" is declared with an initial value of the result of a calculation:

```
var y = 2 + 3
```

In this example, a variable called "name" is declared with an initial value of "John" and a type annotation of "string":

```
var name: string = "John"
```

Assignment Statement

An assignment statement is used to change the value of a previously declared variable. The syntax for an assignment statement consists of an identifier followed by the equals sign and an expression.

> Note that the variable must have been previously declared, and the type of the expression must be compatible with the type declared for the variable

In this example, the variable x is declared with a value of 10, then its value is changed to 5 using an assignment statement:

```
var x = 10
x = 5
print(x)

// Output: 5
```

Function Call Statement

A function call statement is used to invoke a function and execute its code. It consists of the function name followed by parentheses containing the arguments passed to the function. It's important to note that the arguments passed to the function must match the parameters defined in the function's declaration in terms of data type and order.

The example below shows a function called `myFunction` being called with three arguments.

```
myFunction(2, "hello", true)
```

You can find a more detailed explanation of the function call syntax in the Function Call Expression Section.

Function Declaration

Function declaration is used to define a reusable piece of code that can be called multiple times from different parts of the program. It starts with the keyword function, followed by the name of

the function, a set of parentheses containing a list of parameters (if any), and the function's body wrapped in curly braces.

Here is the basic syntax of a function declaration statement:

```
function sum(num1, num2): int {
  var result = 2 + 3
  return result
}
```

Now let's take a look at the three kinds of statements used by function declaration statements in Catscript.

Function Body

The function body is where the actual logic of the function resides. It can contain any valid Catscript statement, such as variable declarations, control structures, and function calls. The function body is defined within the curly braces {} that follow the function declaration statement.

> Note that the function body statement must contain at least one Catscript statement.

The following example shows a valid function body statement.

```
function myFunction(parameter1, parameter2) {
    var x: int = 10;
    if (x > 5) {
        print("x is greater than 5");
    }
    return parameter1 + parameter2;
}
```

Parameter List

The parameter list is a comma-separated list of parameters that the function accepts. Each parameter consists of a name and an optional type annotation. If a type annotation is not provided, Catscript will try to infer the type from the value passed in when the function is called.

In the following example, a parameter list composed of two values with type annotations is shown.

```
function myFunction(parameter1: int, parameter2: bool) {
    // function body
}
```

Return Statement

A function can return a value using the return statement. If the function returns a value, the return type must be specified in the function declaration. The return statement can be used to exit a function early and return a value.

> Note that Catscript expects all branches of code to have a return statement.

The following example shows a function with valid return coverage.

```
function validFunction(parameter1: int, parameter2: int): int {
    return parameter1 + parameter2
}
```

A function with invalid return coverage has at least one branch that will never hit a return statement.

The following example shows a function with invalid return coverage.

```
function invalidFunction(x: int): bool {
    if (x >= 5) {
        return true
    } else {
        print("Less than 5")
    }
}
```

If an argument less than 5 is passed into this function, we will never execute the return statement.

Expressions

Equality Expression

Equality expressions are used to compare two values and determine if they are equal. In Catscript, the `==` operator is used to check if two values are equal, and `!=` is used to check if two values are not equal. Equality expressions evaluate to boolean values and are frequently used as the expression in if statements.

Here is an example that compares two numbers using the equality expression:

```
if (5 == 10) {
    print("x and y are equal")
} else {
    print("x and y are not equal")
}
```

Comparison Expression

Comparison expressions are used to compare two values and determine if one is greater than, less than, or equal to the other. They evaluate to a boolean value and are frequently used in conjunction with if statements. In Catscript, the following comparison operators are available:

`>` : greater than

`<` : less than

`>=` : greater than or equal to

`<=` : less than or equal to

Here is an example that uses the greater than operator to compare two numbers:

```
if (5 > 10) {
  print("x is greater than y")
} else {
  print("x is not greater than y")
}
```

Additive Expression

Additive expressions are used to perform arithmetic operations on numeric values. In Catscript, the addition operator (`+`) is used to add two values, and the subtraction operator (`-`) is used to subtract one value from another.

The following example demonstrates the use of an additive expression in a print statement.

```
print(5 + 10)  // Output: 15
```

Additive expressions can also be used with string values. In this case, the addition operator + is used to concatenate strings together. The example below demonstrates this behavior.

```
var str1 = "Hello"
var str2 = "world!"
print(str1 + ", " + str2) // Output: Hello, world!
```

Factor Expression

A factor expression in Catscript is an expression that involves multiplication or division of two operands. The order of operations applies to factor expressions just like any other mathematical expression. Multiplication and division are performed before addition and subtraction, so it's important to use parentheses when needed to ensure that the expressions are evaluated in the correct order.

The example below demonstrates a simple multiplication between two integers:

```
print(5 * 10) // Output: 50
```

The following is an example of a factor expression that includes parentheses to enforce order of operations with an additive expression:

```
var result = (10 + 2) / 6

print(result) // Output: 2
```

Unary Expression

A unary expression in Catscript is an expression that operates on a single operand. The unary operator can be either `not` or `-`.

The `not` operator performs a logical negation on a boolean value. When applied to a boolean value, it returns the opposite boolean value. This is clear in the example below:

```
var x = true

print(not x) // Output: false
```

The `-` operator performs negation on a numeric value. When applied to a numeric value, it returns the value multiplied by `-1`. Here's an example:

```
var x = 5
print(-x) // Output: -5
```

Primary Expression

A primary expression in Catscript is the simplest form of expression and can take on several forms. It can be an identifier, a literal, a function call, or a parenthesized expression.

Identifiers are used to access the value of a variable or to call a function. In the following example `name` is an identifier.

```
var name = "Alice"
```

Literals are fixed values that are directly represented in the code. There are several types of literals in Catscript, including strings, integers, booleans, null, and list literals.

```
"Hello, world!" // String literal
21              // Integer literal
true            // Boolean literal
null            // Null literal
[1,2,3]         // List literal
```

Function calls are used to invoke a function with zero or more arguments. A function call is denoted by the function name followed by parentheses containing the arguments.

```
function greet(name) {
  print("Hello, " + name + "!")
}

greet("Alice") // Function call
```

Parenthesized expressions are used to group expressions together to enforce order of operations or to clarify code. The expression inside the parentheses can be any valid expression.

```
24 / (10 + 2) // Parenthesized expression
```

List Literal Expression

In Catscript, a list is a collection of ordered values. A list literal is a way to define a list with a specific set of values. List literals are enclosed in square brackets `[]`, with each element separated by a comma `,`.

Here is an example of a list literal containing three integers:

```
[1,2,3]
```

List literals can also include elements of distinct types:

```
[true, "banana", 3, "dog"]
```

Function Call Expression

A function call expression is used to call a function and pass arguments to it. It consists of the function name followed by parentheses containing the arguments passed to the function. The arguments can be expressions that evaluate to the expected data types specified in the function's parameter list.

> Note that the function must be previously defined through a function declaration statement

The syntax for a function call expression is as follows:

```
function_name(argument1, argument2, ..., argumentN)
```

Here is an example of a function call expression that passes two arguments, an integer and a string, to a function called `myFunction`:

```
myFunction(42, "hello")
```

The order and number of the arguments passed to the function must match the parameters defined in the function's declaration. For instance, the function `myFunction` should have a signature like this:

```
func myFunction(num: int, message: string) {
  // function body
}
```

It's important to note that the function call expression is itself an expression, meaning that it evaluates to a value. The value returned by the function can be stored in a variable or used in an expression.

Argument List Expression

An argument list expression in Catscript is used to pass one or more arguments to a function. The syntax consists of zero or more expressions separated by commas, enclosed in parentheses.

The example below shows an argument list used in a function call. The argument list is the series of expressions between the parentheses.

```
myFunction(42, "hello")
```

Type Expression

In CatScript, variables have types that determine the kind of value they can hold. CatScript is statically typed, which means that the variable's type is known at compile-time and cannot be changed during program execution. Type expressions have already been used various times in this documentation. They are used in function declarations, variable statements, and parameter lists. The CatScript type system is relatively small, consisting of the following types:

- `int` : A 32-bit integer.
- `string` : A Java-style string.
- `bool` : A boolean value, which can be either true or false.
- `list` : A list of values with a specific type, denoted as list, where x is the type of the values stored in the list.
- `null` : The null type, which represents the absence of a value.
- `object` : The most general type, which can hold any kind of value.

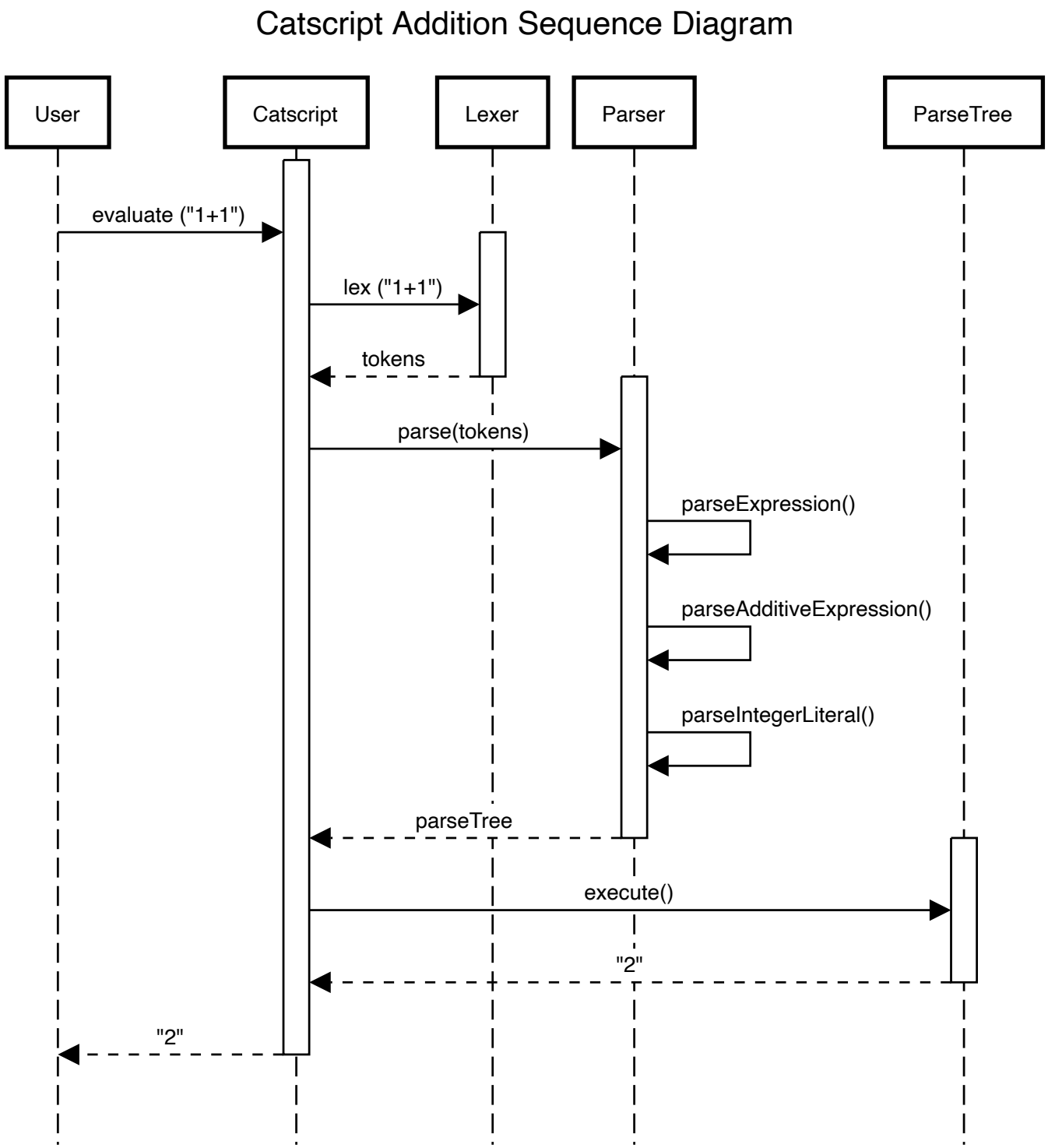
The example below shows a type expression used in a variable statement:

```
var num: int = 42
```

Type expressions are useful for indicating the return type of functions and specifying the parameter types. This is shown in the example below:

```
function isGreaterThan(x: int, y: int): bool {
    return x > y
}
```

Section 5: UML



Section 6: Design trade-offs

For this project, we decided to make a parser by hand instead of using a parser generator. We thought it would be more fun and easier to understand if we used recursive descent parsing. This way, we could really get a better grip on the grammar than if we just used some generator.

By doing recursive descent, we could break the grammar into smaller parts, which made it less confusing and easier to work with. Recursive descent allowed us to see how all the parts of the grammar connected and depended on each other.

So, by making our own parser, we not only got a better understanding of the topic, but also improved our problem-solving and critical thinking skills. It turned out to be a great learning experience that really helped us improve our CS knowledge.

Section 7: Software development life cycle model

In the course of this project, our team implemented the Test Driven Development (TDD) model, which emphasizes writing code to pass a predefined test suite that specifies the expected behavior of the software.

We adhered to the TDD model by constantly refining our code to meet requirements set forth by the tests. This allowed us to identify and fix any issues or bugs early in the development process.

Personally, engaging with the intricacies of the code and experiencing the sense of accomplishment that accompanies the successful completion of each test proved to be highly rewarding. Utilizing the TDD model facilitated not only the enhancement of code quality but also contributed to a more comprehensive understanding of the software development process.