

CSCI 468

Spring 2023

Luke Simonson and Payton Morris

Section 1: Program

A zip file of my source code has been included.

Section 2: Teamwork

For this project, Team member 1 wrote the parser, tokenizer, and compiler while Team member 2 wrote documentation and several tests to ensure the correctness of the parser. The tests that Team Member 2 wrote are as follows and each test passes with the current iteration of the parser:

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

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

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

public class partnerTests extends CatscriptTestBase {
    @Test
    void additiveExpressionInForLoopWorks(){
        assertEquals("2\n4\n6\n", executeProgram("for( x in [1,2,3])\n{print(x+x)}"));
    }

    @Test
    void factorExpressionInForLoopWorks(){
        assertEquals("1\n4\n9\n", executeProgram("for( x in [1,2,3])\n{print(x*x)}"));
    }

    @Test
    void equalityExpressionInForLoopWorks(){
        assertEquals("false\ntrue\nfalse\n", executeProgram("for( x in [1,2,3]){print(x==2)}"));
    }
}
```

Section 3: Design pattern

In the Catscript Parser, there is a function to return the type that a list contains. In this function, we used the memoization pattern because this could be a costly function call with generic types. Memoization allows us to

speed up subsequent calls to this function during type checking and validation. The following code is located in CatscriptType.java starting 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.

Introduction

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

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

Datatypes

There are five datatypes recognized in Catscript: integers, strings, booleans, list and object types. Null values can be stored as any datatype in Catscript. Datatypes can be defined either explicitly or implicitly, as shown below.

Explicit definition:

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

Implicit definition:

```
var x = 5
```

Integers

Catscript Integers are whole number values stored as 32-bit unsigned integers. They can be explicitly defined in variables using the "int" keyword.

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

Strings

Catscript strings are collections of characters enclosed by double quotes. They are defined explicitly using the "string" keyword, or implicitly as follows:

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

Or

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

Booleans

Booleans in Catscript contain either "true", "false", or "null". They can be defined using the "bool" keyword as follows:

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

Lists

Catscript lists are collections of variable lengths of values. Lists can be defined by square brackets enclosing comma separated values. Lists can be of any one (and only one) type, and the type can be specified by enclosing it in left and right angle brackets.

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

Objects

In Catscript, objects are the wildcard of types. Objects can store any of the datatypes previously mentioned and are defined using the object keyword.

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

Variables

Variables in Catscript are denoted by the keyword "Var" followed by the variable name. The name must start with a letter and may contain only letters and numbers. They are initialized and defined as seen above and can be followed by an explicit type expression.

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

Or

```
var y = "hello world"
```

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

```
var x = 5  
x = 6
```

Comments

Comments are not read by the compiler. They are defined by using two slashes in a row. Each line of comments needs its own double slash.

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

Expressions

There are eight types of expressions in Catscript, made up of variables or literals as well as operators. The operators are typically infix except for "not" and "!" and type expressions.

Primary Expressions

In Catscript, a primary expression consists of any of the literal expressions for datatypes, as well as identifiers. Primary expressions can be enclosed in parentheses.

```
x
y
"hello world"
null
false
true
```

```
(true)
[1,2,3]
```

Unary Expressions

There are two unary operators in Catscript, "-" and "not". The "not" operator will negate a boolean value, while the "-" operator will negate a number value. Unary expressions consist of at least one unary operator, followed by another or a primary expression, as follows:

```
-5
-(-5)
not true
```

Factor Expressions

Factor Expressions in Catscript are used to divide and multiply two Primary expressions. The operators used are "*" and "/"

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

Additive Expressions

In Catscript, Additive expressions are used to add or subtract two values. Two integers may be added or subtracted, Strings may be concatenated to other strings, and an integer may be converted to a string and concatenated as well. Examples are as follows:

```
x+y
y-x
"hello" + "world"
"hello" + 1
"hello" + null
```

Comparison Expressions

Comparison expressions are used to compare two expressions, using four operators to compare the lefthand side to the righthand side. Comparison expressions evaluate to boolean values such as "true" and "false". The four available operators for comparison expressions are the less than operator ("<"), greater than (">"), less than or equal ("<="), and greater than or equal (">=").

```
x < y  
x <= y
```

```
x > y  
x >= y
```

Equality Expressions

Equality expressions in Catscript will determine if the lefthand side is equal to the righthand side. Using two operators, the equality operator ("==") and the not-equals operator ("!="), the equality expression will return a boolean value "true" or "false".

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

Statements

In Catscript, statements are phrases made up of expressions, keywords, datatypes, and other symbols. Statements can mutate variables, call functions, return variables and produce output. Of the six types of statements, four that have not been covered will be shown below, with the inclusion of function declarations and bodies, due to their containment of statements and expressions.

Print Statement

In Catscript, using the "print" keyword followed by an expression enclosed in parentheses will invoke the print statement. This will print the value returned by the expression to the console, or other explicitly defined output area.


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

If statement

If statements in Catscript use a comparison expression to determine if a block of statements should be run. The statements start with the "if" keyword and are followed by a comparison expression enclosed in parentheses. If the comparison statement returns "true", the block of code will run. Optionally, the if statement can be followed by an "else" statement, defined by the "else" keyword. If the if statement returns "false", the code below the "else" statement will run before continuing on.

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

For Loops

For loops in Catscript are an iterative loop starting with the keyword "for" and followed by an identifier, "in", and another identifier that evaluates to a list or a list literal, all enclosed in parentheses. Within the for loop, the first identifier inside the parentheses will be assigned to the next value in the list for each iteration of the loop. Following the loop, there is a block of code enclosed in curly brackets which will evaluate on each iteration.

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

Functions and Function Declarations

Functions in Catscript are declared using the "function" keyword followed by an identifier for the function and closed parentheses. A set of parameters for the function can be set in the parentheses, which can be given explicit types during declaration, similar to variables. After the parameters, there is a block of statements enclosed in curly brackets called the function body. The body is executed when the function is called.

Functions may have one of five 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.

```
function foo(x: int) {  
    print(x)  
}  
foo(5) //prints 5
```

Function Call Statement

A function call statement in Catscript calls a function using the identifier of the function. If the function requires parameters, these are set in the function call statement parentheses. A function call statement can also be passed to a variable, assuming the return type is not void.

```
foo(5)  
x = foo(5)
```

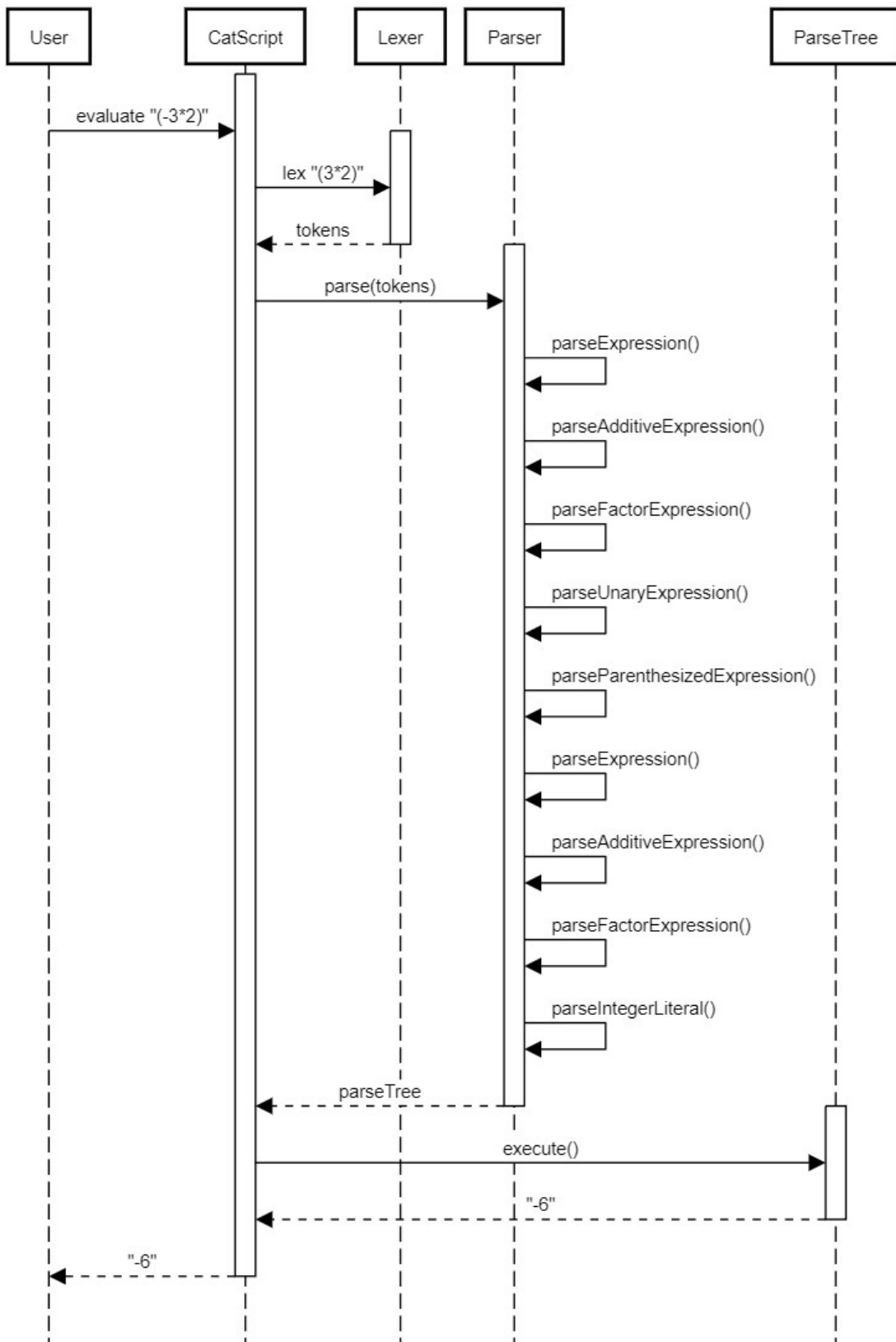
Return Statement

Return statements in Catscript can only be called within a function body. Denoted by the keyword "return" and followed by an expression (similar to print statements), the value from the evaluated expression will be returned by the function itself.

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

Section 5: UML.

Catscript Parenthesized Multiplication Sequence Diagram



Section 6: Design trade-offs

We used the recursive decent algorithm during this class to write a parser by hand rather than using a parser generator. While this took more manual effort, it also gave us a much better understanding of how parsers function as well as being much easier to debug if and when bugs were identified.

Section 7: Software development life cycle model

Throughout this project, we used Test Driven Development. This helped our team by allowing us to immediately ensure that our tokenizer, parser, and compiler were correct and had minimal bugs. The one downside I experienced using this method is that a good chunk of our time was devoted to running tests in between making changes to the code, though that would be the case without having unit tests written as well in some ways. The test driven nature of this project also allowed us to ensure that we did not miss implementing anything as the test suite would fail if we did so.