

Section 1:

<https://github.com/erinscheunemann/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

Section 2: For teamwork, we worked in a two-man group. My partner, Silas Almgren, was tasked with quality assurance and documentation while I was tasked with developing our compiler. My partner provided three high-level tests for me to use to test the compiler with. These three tests were: `nestedIfExecutes`, `typeCastsWork`, and `throwsException`. (these tests are located in the `testsfrompartner.java` file in the `/eval` folder in the `/test` directory of the compiler) These were written to capture parts of the compiler that weren't included in the many tests already provided to us. They also provided me with a 4-page document that detailed how the CatScript programming language works. I was in charge of development so I was in charge of writing the code for our compiler. I was in charge of making sure the tests we were provided with and the ones that my partner wrote passed.

Tests from partner:

```
1 package edu.montana.csci.csci468.eval;
2
3 import edu.montana.csci.csci468.CatscriptTestBase;
4 import org.junit.jupiter.api.Test;
5 import edu.montana.csci.csci468.parser.ParseException;
6
7 import static org.junit.jupiter.api.Assertions.*;
8
9 public class testsfrompartner extends CatscriptTestBase {
10     @Test
11     void nestedIfExecutes() {
12         assertEquals("small\n", executeProgram("var enabled = true\n" +
13             "var x = 2\n" +
14             "if(enabled) {\n" +
15             "    if(x < 5) {\n" +
16             "        print(\"small\")\n" +
17             "    }\n" +
18             "});"));
19     }
20
21     @Test
22     void typeCastsWork() {
23         assertEquals("2\n", executeProgram("var x: int = 2\n" +
24             "var y: object = x\n" +
25             "print(y)"));
26         assertEquals("hello\n", executeProgram("var x: string = \"hello\"\n" +
27             "var y: object = x\n" +
28             "print(y)"));
29         assertEquals("null\n", executeProgram("var x = null\n" +
30             "var y: object = x\n" +
31             "print(y)"));
32     }
33
34     @Test
35     void throwsExceptions() {
36         assertThrows(ParseErrorException.class, () -> executeProgram("var x = [1,2]));
37         assertThrows(ParseErrorException.class, () -> executeProgram("var x = \"hi\"\n" +
38             "var x = \"hello\""));
39         assertThrows(ParseErrorException.class, () -> executeProgram("var x: int = \"hello\""));
40     }
41 }
42
43 }
```

Section 3: Our parser uses memoization within the `getListType` function in the CatScript parser. Originally the `getListType` function was creating a new `ListType` object every time the function was called. This could be very costly down the line as creation of a new object is resource inefficient. Memoization allows for us to store the different `ListType` objects corresponding to different types as they are created in a cache. In our case this cache is a map object. This means that instead of creating a new `ListType` object every time, we can first check to see if the object corresponding to the component type has already been created and if it has then we return that already created item and if it hasn't then we create it and add it to the hash map with its component type as the key.

Section 4:

Catscript Guide

A language reference for Catscript

CSCI468

May 2023

Introduction

This document is a reference guide for the Catscript language. Catscript is a simple, statically typed scripting language. It can be compiled to Java bytecode to be executed on the JVM. Here is an example of a simple Catscript program:

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

Catscript programs can be written and executed with the `CatScriptServer` web utility packaged with the source code. Additionally, you can write Catscript programs in your favorite editor, save them with the `.cat` extension and compile them directly to Java bytecode.

Features

Variables

Variables in Catscript are declared using the `var` keyword. Variables must be initialized to a value in order to be syntactically valid. Variables in Catscript are statically typed but the language does have type inference. This means you can optionally give your variables explicit types when they're declared. Here are two examples of valid variable declarations:

```
var x = 10
```

```
var y: string = "hello"
```

```
print(x)
```

```
print(y)
```

```
> '10'
```

```
> 'hello'
```

Variable Type System

Catscript is statically typed and adheres to the following minimalist type system:

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list - a list of value with the type 'x'
- null - the null type
- object - any type of value

Print statement

The `print()` statement is used to display a value to the console. Any valid expression can be placed in this statement - such as variables, literal values, and function calls.

```
var hi = "Hello, world"
```

```
print(hi)
```

```
> 'Hello, world'
```

Operators

Catscript has four basic mathematical operators: addition `+`, subtraction `-`, multiplication `*`, and division `/`. These operators can all be used to perform mathematics on integer literals and variables of the int type. The `-` keyword for subtraction also works for signing integers (i.e. setting negative values). Additionally, the addition `+` operator can be used for string concatenation. Finally, there is also the `not` operator which can be used on booleans to flip their value.

```
var x = 1 + 1
```

```
print(x)
```

```
> '2'
```

```
var y = x * 2
```

```
print(y)
> '4'

var z = -1

print(z)
> '-1'

var hi : string = "Hello, "

var name = "Bob"

print(hi + name)
> 'Hello, Bob'

print(not true)
> 'false'
```

Comparisons

Catscript also supports the following comparisons between integer values: less than `<`, greater than `>`, less than or equal to `<=`, greater than or equal to `>=`, equal to `==`, and not equal to `!=`. Comparisons will evaluate to booleans.

```
10 > 5
> 'true'

3 <= 6
> 'true'

x = 1
y = 2
x == y
> 'false'
```

Conditional statements

Conditional statements can be used to optionally execute code based on the result of a boolean expression. This is done by using the `if` keyword followed by a boolean expression wrapped in

parentheses - the code to conditionally execute then follows wrapped in curly braces. For example:

```
x = 12

if (x > 10) {

    print("that is a big number")

}

> 'that is a big number'
```

Additionally, the **else** keyword can be used to denote a block of code to run if the boolean expression evaluates to false. For example:

```
x = 5

if (x > 10) {

    print("that is a big number")

} else {

    print("that is a small number")

}

> 'that is a small number'
```

In addition to plain **else** statements, **else if** statements can be used to apply additional logic after an initial if statement. For example:

```
x = 2

if (x == 1) {

    print("doing action 1")

} else if (x == 2) {

    print("doing action 2")

} else if (x == 3) {

    print("doing action 3")

}
```

```
> 'doing action 2'
```

For loops

In Catscript, for loops can be used to iterate over lists. When a for loop is executed, it will iterate over the items of the specified list, storing these values in a variable of your choosing. Each iteration, this variable will update and the body of the loop will execute. For example:

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

```
  print(x)
```

```
}
```

```
> '1'
```

```
> '2'
```

```
> '3'
```

Functions

In Catscript, functions can be used to define a block of code that you would like to reuse in your code. Functions may return a value with a specified variable type or may simply be **void** and return nothing at all. Additionally, functions can be passed any number of parameters to be used in their execution. These parameters are optionally typed, unless you are performing type specific operations. See below.

```
function printer(x) {
```

```
  print(x)
```

```
}
```

```
printer("hi")
```

```
> 'hi'
```

```
function add(x: int, y : int) {
```

```
  return x + y
```

```
}
```

```
var num = add(1,2)
```

```
print(num)
```

```
> '3'
```

It is also worth noting that Catscript functions support recursion.

Errors

Incompatible Types

When you encounter this error you will see the message: **Incompatible types**. This occurs when when you attempt to assign data to a variable of a conflicting type or when you attempt to perform an operation on a data type that isn't supported. For example:

```
var x : int = "example"
```

```
> 'Error: Incompatible types'
```

Unterminated List Literal

Indicated by the error message: **Unterminated list literal**, this error occurs when the closing bracket `]` is omitted when initializing a list.

```
var x = [1,2,3
```

```
> 'Error: Unterminated list literal'
```

Unterminated Argument List

This error occurs when a function is invoked with a dangling comma in its list of arguments. The error message: **Unterminated argument list** will be shown.

```
function test(x,y) {  
  print(x + y)  
}
```

```
test(1,)
```

```
> 'Error: Unterminated argument list'
```

Duplicate Name

This error occurs when a variable is initialized with a name that is already used in the current scope. The error message: **This name is already used in this program** is shown.

```
var x = 1
```

```
var y = "test"
```

```
> 'Error: This name is already used in this program'
```

Unknown Name

This error occurs when a variable name is referenced that doesn't exist. The error message: **This symbol is not defined.**

```
print(x)
```

```
> 'Error: This symbol is not defined'
```

Argument Mismatch

This error occurs when a function is invoked with an incorrect number of arguments. The error message: **Wrong number of arguments is shown.**

```
function add(x:int,y:int): int {  
    return x + y  
}
```

```
add(1)
```

```
> 'Error: Wrong number of arguments'
```

Missing Return Statement

This error occurs when a function with an explicit return type doesn't contain a return statement or if a function only has partial return coverage. The following error message is shown:

Missing return statement in function

```
function test(x,y) : int {  
    if(x > y) {return x}  
}
```

```
> 'Error: Missing return statement in function'
```

Unexpected Token

This is a general syntax error that occurs when you have a character somewhere it shouldn't be or if you are missing a token somewhere in an expression or statement. The error message:

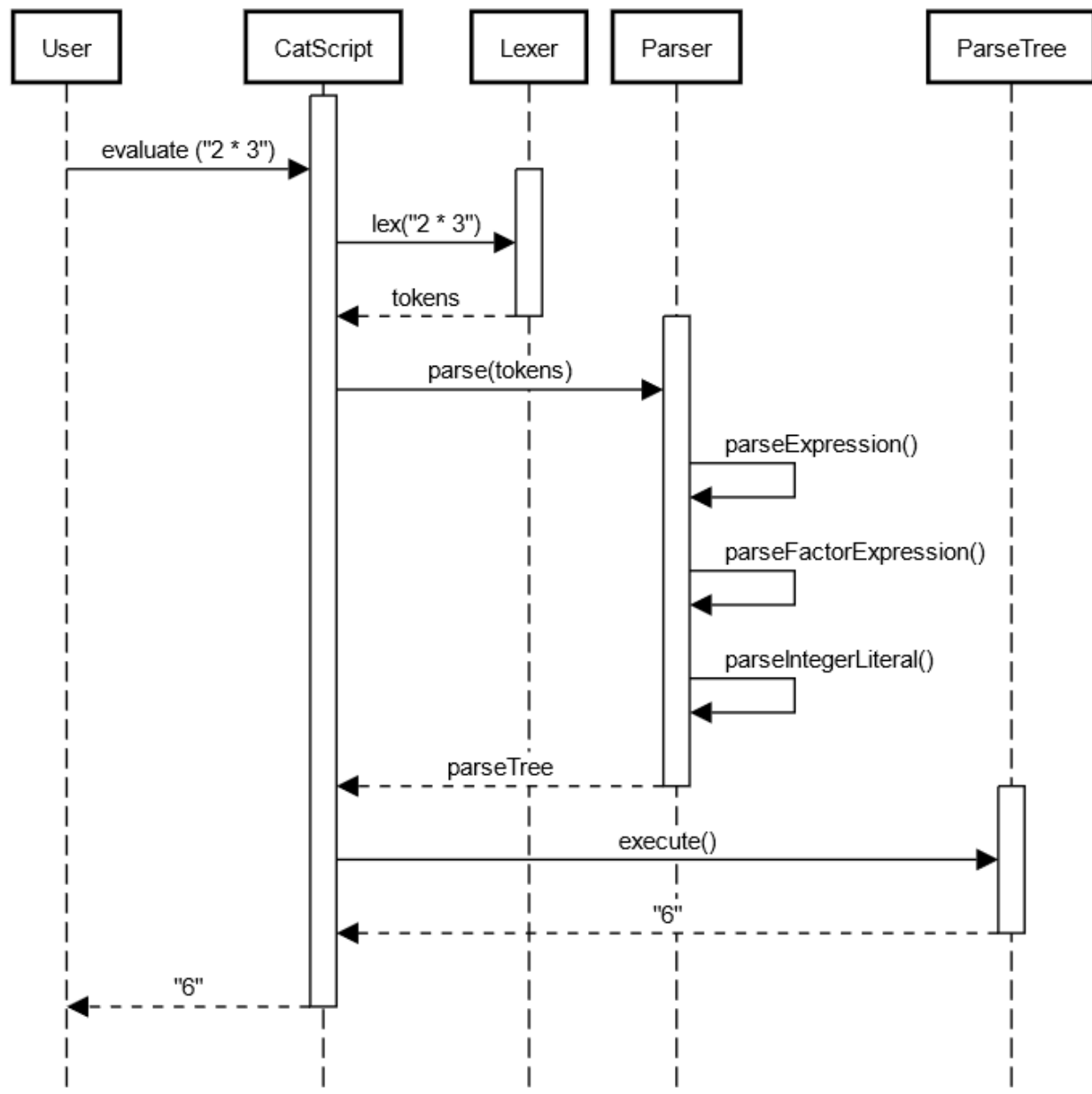
Unexpected token is shown. For example, if the **=** is missing from a variable assignment statement.

```
var m "j"
```

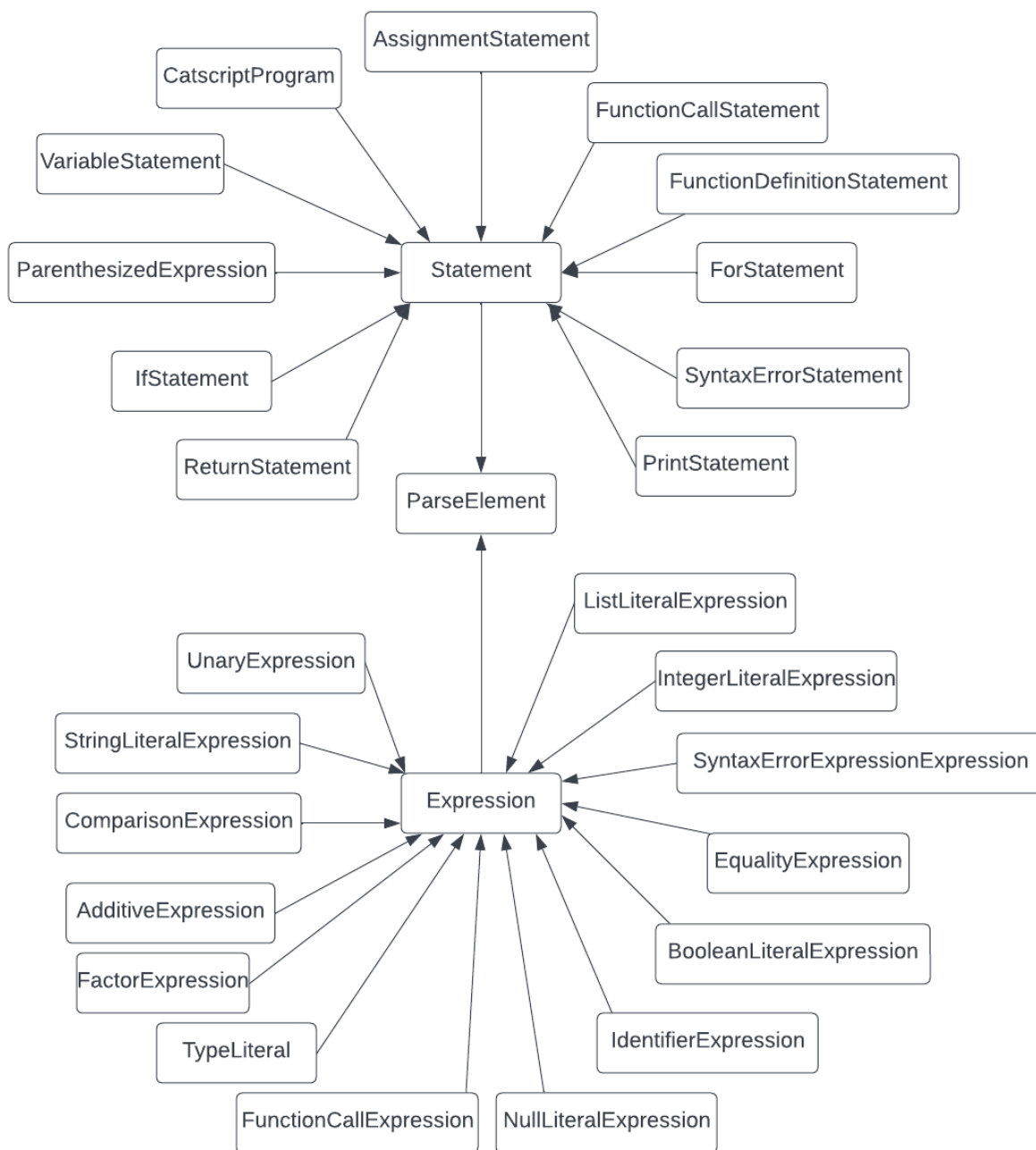

> 'Error: Unexpected Token'

Section 5:

Catscript Multiplication Sequence Diagram



This is a sequence diagram detailing the process that the CatScript compiler goes through to tokenize, parse, and execute the command "2*3". This helps to show the process and how everything is connected within the compiler. It is shown that the string is broken into lexemes by the lexer, or tokenizer, and then those tokens are sent to the parser which returns a parse tree. This parse tree is then used to execute the command and returns the result, 6, to the user.



This is a class diagram detailing how all the classes work together in the compiler. There are two main parent classes, Expression and Statement, from which the various expressions and statements within the language inherit from. These are then used by ParseElement to make up the compiler's parsing and execution functionalities.

Section 6: While designing our parser we had many design choices to make in order to make our parser work effectively and efficiently. One of the choices we made was to not use the visitor pattern that is common in many other compilers. The visitor pattern abstracts the operators or behaviors from the objects that perform these behaviors. This would have allowed us to write

visitors that would have accepted objects of our classes, in this case expressions or statements, and allowed us to perform operations on them. Instead we chose to add those functions directly to the classes. This meant that we would have to add an execute and compile method to each expression or statement class but it saved us a lot of complexity that we would have added had we used the visitor pattern. This allowed us to better focus on the code we were writing rather than trying to figure out how to use one design pattern over a multitude of classes.

Section 7: For our software development life cycle (SDLC) we went with test driven development (TDD). TDD is a SDLC where software is written in order to pass tests that were written using the project's specifications. We choose to go with this SDLC over others because we are starting with very specific specifications from the CatScript grammar so it made sense to turn these specifications into tests and then use those to figure out what code to write. We started with a number of tests for each part of the CatScript compiler starting with the tokenizer and then moving on to the parser and so on. This allowed for us to write code for the compiler in an order that made sense and allowed for us to check whether or not our code was to the set specifications along the way.