

Catscript Compiler

Alexander Fischer
Senior Capstone

5/5/2023

—

CSCI 468

—

Mr. Carson Gross

Introduction

This project serves as a demonstration of a compiler constructed for Montana State University's proprietary programming language, Catscript. The scope of work involved the creation and implementation of solutions pertaining to Tokenization, Parsing, Evaluation, and Bytecode. Furthermore, a Testing Engineer was responsible for assessing the compiler's performance through a sequence of customized unit tests.



Gianforte School of Computing

Program:

The entire project source code can be found on GitHub:
<https://github.com/alex-fisch/csci-468-spring2023-private/tree/main/src>

Teamwork:

This project was divided into two separate roles: Software Engineering and Software Testing Engineering. The software Engineer was responsible for implementing working solutions for Tokenization, Parsing, Evaluation, and Bytecode. The Testing Engineer created unit tests to test the compilers functionality.

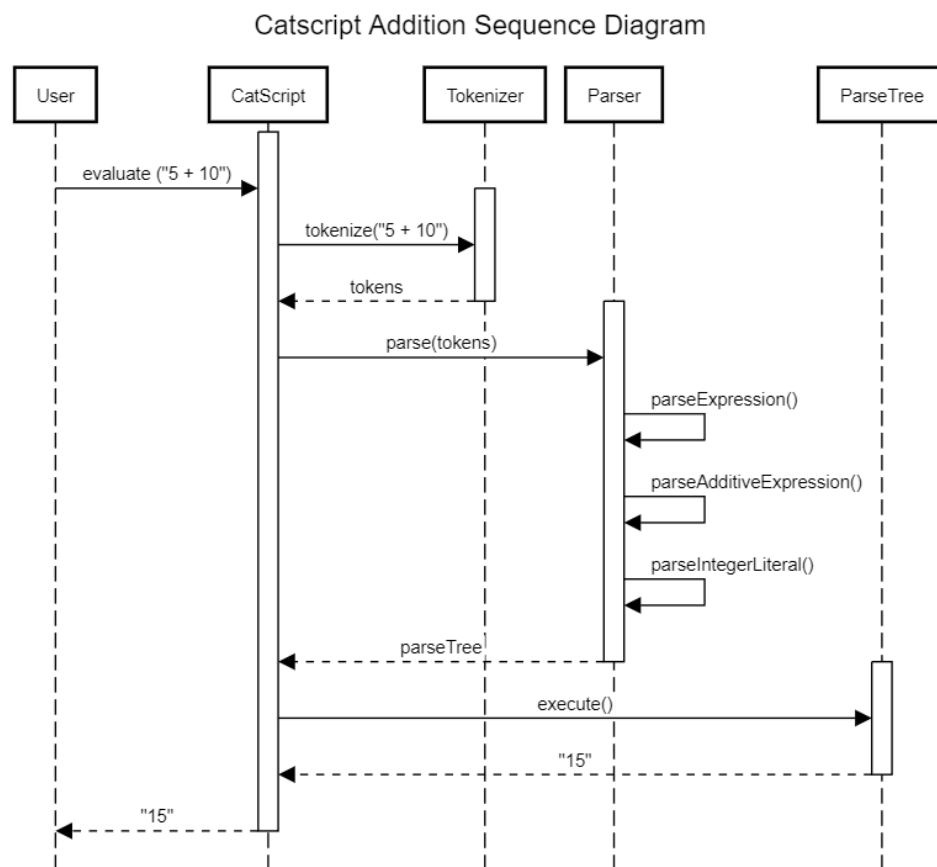
Alexander Fischer

- Software Engineer
- Est. 150 hours

Devan Eastman-Pittman

- Testing Engineer
- Est. 20 hours

UML sequence diagram that describes the addition function of the compiler



Design Pattern

In the Catscript scripting language, the design pattern used is Memoization. It is used to essentially optimize the `getListType` method in the `CatscriptType` class. Memoization is a pattern that aims to eliminate redundant operations that would cause a worse time complexity by remembering the results of a function or process. The `getListType` method uses a `ConcurrentHashMap` called `LIST_TYPE_CACHE` to store cached `ListType` objects. When the method is called with a `CatscriptType` input, it will first check the cache for a matching `ListType` object. If the object is not found, the method creates a new `ListType` object and stores it in the cache for future use. Subsequent calls to the method with the same input will return the cached `ListType` object, eliminating the need for further computation and improving runtime speed. By reducing the creation of duplicate objects, this implementation also reduces memory cost.

This is located in `src/main/java/edu.montana.csci.csci468/parser/CatscriptType.java` at line 35-40

```
35 // TODO memoize this call
36 1 usage
37 static final ConcurrentHashMap<CatscriptType, ListType> LIST_TYPE_CACHE = new ConcurrentHashMap<>();
38 // This line creates a ConcurrentHashMap to store the cached ListType objects.
39 5 usages new *
40 public static CatscriptType getListType(CatscriptType type) { //This line tries to get the ListType object from the cache by reading the given CatscriptType
41     return LIST_TYPE_CACHE.computeIfAbsent(type, ListType::new); //If the ListType object is not already in the cache, computeIfAbsent will create one
42 }
```

Design Trade-offs

There were two main design considerations for this project. The first one involved choosing either Recursive Descent or a Parser Generator. The second design trade-off was about balancing the specificity of the separation of concerns with simplicity. Recursive Descent was ultimately selected for this project because it closely aligns with the interpretation of the language's grammar. Because of this the project required longer and more detailed procedures. While designing the codebase for this project, it was important to carefully consider the various components of an application that is being designed. One method used to achieve this is called the separation of concerns, where certain procedures are tightly coupled within the application itself. This provides modularity and defined scopes for the components to operate within the application. For this project, however, simplicity and code location were prioritized over the separation of concerns. The evaluation and compilation procedures were directly connected to the parse tree nodes, which is typically unusual because these procedures have different concerns. However, the main goal with this trade-off was to simplify components within the codebase.

Custom Unit Tests

The testing engineer wrote a set of custom unit tests for this project. They are listed below.

```
1 package edu.montana.csci.csci468;
2
3 import edu.montana.csci.csci468.tokenizer.Token;
4 import org.junit.jupiter.api.Test;
5
6 import java.util.List;
7
8 import static edu.montana.csci.csci468.tokenizer.TokenType.*;
9 import static org.junit.jupiter.api.Assertions.assertEquals;
10
11
12 public class CapstoneTestsAF extends CatscriptTestBase {
13
14     @Test
15     void test1() {
16         assertEquals("20\n", compile("20"));
17
18         assertEquals("false\n", compile("false"));
19
20         assertEquals("100\n", compile("20 * 5"));
21     }
22
23
24
25
26     @Test
27     void test2() {
28         assertEquals("37\n", compile("function foo(x) { print(x) }" + "foo(37)"));
29
30         assertEquals("[9, 57, 2]\n", compile("function foo(x : list) { print(x) }" + "foo([9, 57, 2])"));
31
32         assertEquals("21\n", compile("var y = 21\n" + "function foo() {}" + "foo()" + "print(y)"));
33     }
34
35
36
37     @Test
38     void test3() {
39         assertEquals("orange\n49\n", compile("print(orange)\n" + "print(49)"));
40
41         assertEquals("99\n", compile("var x = 99\n" + "var y = x*n" + "print(y)"));
42
43         assertEquals("2\n3\n", compile("for(x in [1, 2, 3]) { if(x!=1){print(x)} }"));
44     }
45
```

Section 4 Documentation:

Catscript Language Documentation:

Catscript is a statically typed language based on java and compiled using a JVM and is designed for the compiler we worked on.

Variable Types:

Catscript supports the following variable types for use:

- int for integers
- string - a java style string
- bool - for boolean values
- list for initiating a list of values
- null for null values
- object to create an instance of an object

To create an instance with a given type you could type, **String string_name = "name";** or **var x = 17;**. The first is an explicit type declaration for that variable while the latter is a non explicit one where the type is inferred based on context clues using the var keyword.

Arithmetic Operations:

In catscript you can perform all basic arithmetic operations in your code.

To use **addition** you can use the **“+”** symbol,

For **subtraction** the **“-”** symbol,

For **multiplication** the **“*”** symbol,

For **division** the **“/”** symbol.

Below are a few examples of Arithmetic operations in catscript:

```
int my_num;  
my_num = 1+1;  
System.out.println(my_num); //prints out 2  
  
int my_num;  
my_num = 1-1;  
System.out.println(my_num); //prints out 0  
  
int my_num;  
my_num = 3*2;  
System.out.println(my_num); //prints out 6  
  
int my_num;  
my_num = 8/2;  
System.out.println(my_num); //prints out 4
```

Comparison And Equality Operations:

To compare two pieces of data we have multiple options in catscript, the less than operator "<", the greater than operator ">", the less than or equal operator "<=", the greater than or equal operator ">=", and finally the equals operator "==". These are checked from left to right meaning, is the left hand value "comparison operator here" the right hand value and this will return a bool of true or false.

```
System.out.println(1<2); //prints true  
System.out.println(2<1); //prints false  
  
System.out.println(2<=2); //prints true  
System.out.println(3<=2); //prints false  
  
System.out.println(2>1); //prints true  
System.out.println(1>2); //prints false  
  
System.out.println(2>=2); //prints true  
System.out.println(1>=2); //prints false  
  
System.out.println(1==1); //prints true  
System.out.println(1==2); //prints false
```


Unary Expressions:

You can use the “not” keyword or “-” symbol. Not can be used for getting the opposite of a bool type variable while - can be used on integer type values.

```
int a = 8
int b = -3 print(x * y) //prints -24 correctly instead of 24
var c = false
System.out.println(not c) //prints true
```

For Loops:

For loops in catscript are used for iteration over a fixed period, either to continually execute a set of code or to go through a subscriptable data object.

```
a=[2,4,8];
for(x in a){
  System.out.println(x); // prints 2 4 8
}
```

While Loop:

While loops are used to continually run a set of code while a certain condition is true, if false the loop ends. While loops are useful when you have a variable amount of times you wish to run code. You can complete anything a while loop can do with a for loop and vice versa.

```
x=0;
while(x<3){
  System.out.println(x); // prints 0 1 2
  x++;
}
```

Print Statement:

```
print_statement = 'print', '(', expression, ')'
```

The print statement will bring the expression

Functions:

Primary Expressions:

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false"  
| "null" | list_literal | function_call | "(", expression, ")"
```

Primary expressions are the default configuration for expression parsing in a given

Software Development Lifecycle

The Catscript Compiler project was made through a series of various development stages. For each category there was a separate java test suite that needed to pass successfully. This showed that the software system was functional and achieved its goals. The different test categories that were used are as follows: Tokenization, Parsing, Evaluation, and lastly Bytecode. Each test suite had to be completed to allow the next one to pass. This made the project dependent on everything to move on.