Compilers: CSCI 468

Capstone CatScript Compiler Portfolio

May 2024

by Matthew A. Phillips

Team Members:
Jared Matury

Section 1: Program:

To satisfy the first section of the capstone project is implementing the code for the Catscript programming language. This has been completed and may be seen within the zipped source file included with the submission. All of the files required to run and test the compiler are included within this source.

Section 2: Teamwork:

In section two of the capstone project we are required to work with a partner where that other partner writes the documentation. To this end I worked with Jared Matury who will be referred to as team member 1. I completed the implementation and coding of the compiler, and he wrote all of the documentation that covers in-depth concepts about how the language functions as well as provides examples of code to demonstrate those concepts.

For implementation of the compiler, first came the tokenizer in order to break apart the input code to lexemes and then those are grouped together based on context for the creation of lexical tokens. These tokens can then be parsed, but another important note about tokenizing is through the process of lexical scanning, which locates patterns to be grouped into the lexical tokens, syntactical errors can be found and reported to the user. The resulting tokens are a collection of the following types: syntax tokens, literal tokens, and keywords. All of which are defined in the `TokenType` class.

After the tokenizer came the parser where the tokens were parsed into Expressions and Statements that can be compiled to Java Virtual Machine (JVM) Bytecode. These Expressions and Statements are the fundamental makeup of the language in how they are structured to work together. In parsing them we used a recursive descent method that allowed for the abstraction of most ambiguity. Expressions evaluate to values while Statements execute, or rather they are the running instructions that complete operations on the values of Expressions.

Once everything has been parsed, then we can compile everything into bytecode for execution by the JVM. This, in turn, creates an executable binary for computers to read and run. This step included writing compilation methods for all the Statements and Expressions which took their values and operations and turned them into bytecode, similar to assembly, that represented them in a way that the JVM could understand.

To verify this process was fully operational, team member 1 also wrote tests to determine the full functionality of the language, the following is the test written by team member 1 for the code:

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

import edu.montana.csci.csci468.CatscriptTestBase;
import edu.montana.csci.csci468.parser.expressions.*;
import edu.montana.csci.csci468.parser.statements.*;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class PartnerTest extends CatscriptTestBase {
      @Test
      public void functionCallStatementWithMixedArgumentTypes() {
      assertEquals("The factorial of 5 is: 120\n", executeProgram(
            "function factorial(n : int) : int {\n" +
                        "     if (n == 0) {\n" +
                        "     return 1\n" +
                        "     }\n" +
                        "     else {\n" +
                        "     n = n * factorial(n - 1)  " +
                        "     return n \n" +
                        "     }\n" +
                        "}\n" +
                        "var result = factorial(5)\n" +
                        "print('The factorial of 5 is: ' + result)\n"));
      }

      @Test
      public void FunctionAdditionWorks() {
      assertEquals("The result is: 15\n",executeProgram(
            "function sum( a : int,  b : int) : object {\n" +
            "var c =  a + b\n" +
            "return c }\n" +
            "var result = sum(7, 8)\n" +
            "print('The result is: ' + result) "));

      }

      @Test
      public void nestedForLoopStatement() {
      assertEquals("5\n6\n7\n6\n7\n8\n7\n8\n9\n", executeProgram(
            "for(x in [1, 2, 3]){ for(y in [4, 5, 6]){ print(x + y) } }"));
      }
}
```

The time spent implementing and writing the compiler is estimated to have been 40-50 hours, and the estimated time completion of writing the documentation and tests is 5-10 hours. The following table contains a breakout of the specific hour estimations:

Task	Hours Spent
Implementing Tokenization	3
Implementing Parsing Expressions	6
Implementing Parsing Statements	17
Implementing Bytecode Compilation	18
Writing CatScript Tests	2
Writing CatScript Documentation	5

Section 3: Design pattern

Within the code that was implemented, the design pattern utilized was memoization. Memoization is the utilization of hashmaps to cache data, in this case list types, for quicker fetching during code compilation. Below is a screenshot of the code implementing Memoization, and in the code is sectioned off by comments (which is also pictured).

```
// ------------------------------------------------------------
// Memoized Function
private static final Map<CatscriptType, CatscriptType> LISTS = new HashMap<CatscriptType, CatscriptType>();
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType lType = LISTS.get(type);
    if(lType == null){
        lType = new ListType(type);
        LISTS.put(type, lType);
    }
    return lType;
}
// ------------------------------------------------------------
```

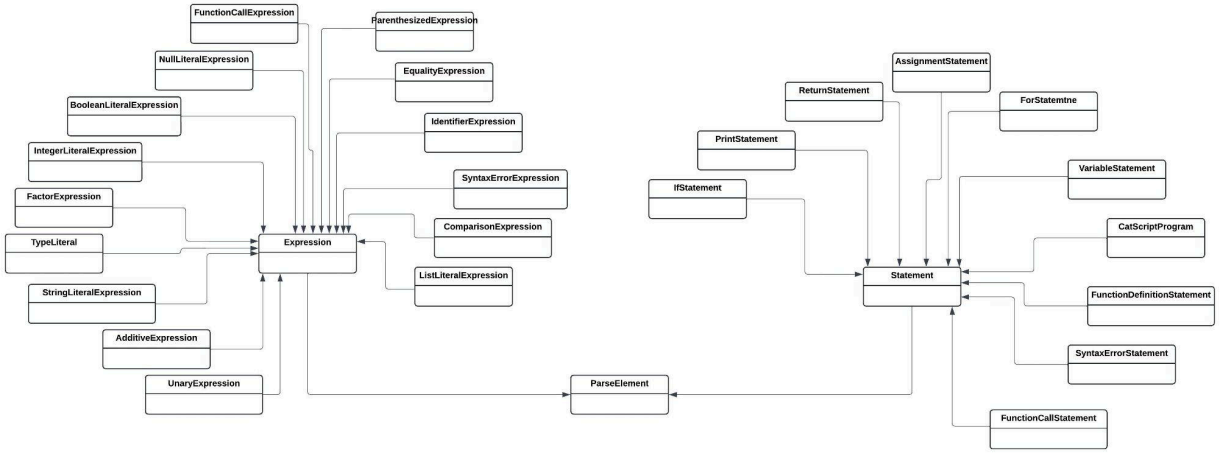
Section 4: Technical writing

The technical explanation of CatScript functionality was written by partner 1 and is appended at the end of this document.

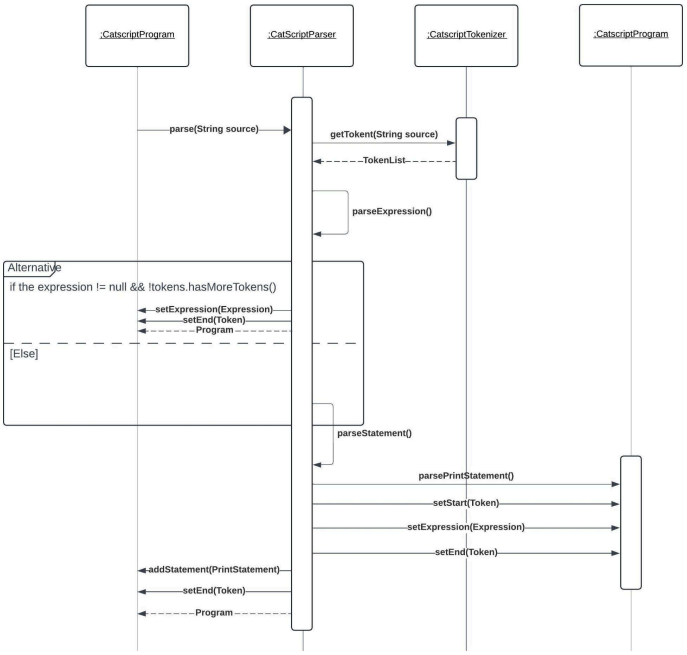
Section 5: UML

To begin with implementation; it is important to be able to understand the relationships between the classes, so included below is a class diagram to show these relationships. It depicts how Statements and Expressions both stem from an object named `ParseElement`; while all other objects stem from either the `Expression` object class or the `Statement` object class. The

classes that extend from the `Expression` class are all of the possible expressions that can be parsed by CatScript and same goes for all of the classes extending the `Statement` Class; which is that all extensions represent the possible statements possible to be parsed by CatScript. Another important note for an extension of the Statements class is the `CatScriptProgram` class which is where the program starts. This is an extension of the statement class since it does not evaluate to a value, but instead an environment; kind of like how each function creates a scope save for the fact that all function scopes are contained within the `CatScriptProgram`.



To properly understand the process used in parsing, below is a sequence diagram that depicts the parsing of a `PringStatement`. This should help provide insight into the required operations to parse an element within the CatScript Language.



Section 6: Design trade-offs

With CatScript, as mentioned previously, we used a recursive descent parser in contrast to a parser generator. The primary drawback is the time required to write recursive descent far exceeds that of a parser generator. The amount of knowledge of parser operations required is also far greater by going the recursive descent route; however, since writing the compiler was for educational purposes this actually makes it a better option. Using recursive descent also does not allow for clear grammar rule definitions; instead we needed to define the grammar before setting into writing it. This probably would need to be done either way, but with recursive descent it is essentially required to keep a clear vision of the desired language grammar in mind while coding the implementation.

Overall the benefits of recursive descent, for the sake of educational purposes, far outweigh the benefits of using a parser generator. It achieves a much deeper understanding of the intimate execution in parsing the lexical tokens. Furthermore, for debugging errors in implementation the recursive descent algorithm allows its writer to filter through the different operations ongoing to determine what and where the error is occurring.

Section 7: Software development life cycle model

For the development of CatScript we used a Test Driven Development (TDD) model which allowed us to set measurable progress in the form of checkpoints. These checkpoints allowed the gradual progression of implementation from tokenizer to parser then to evaluation and finally the compilation to bytecode. The tests also allowed for us to know concretely whether the source was properly being handled by the tokenizer, parser, and compiler; thus ensuring achievement of the desired behavior.

On the level of enjoyment aspect, this was a very enjoyable model to follow. The satisfaction from getting a suite of tests passing and seeing all the green checkmarks light up is hard to describe properly. Furthermore, in relation to academics, it helps reassure me as the student that I know what I am doing in relation to implementing what is learned from lectures. The tests helped make certain I did not stray too far from the intended path, due to misconceptions or other reasons, and usually I was able to clear up the misconceptions on my own by looking farther down the development pipeline.

CatScript Guide

by Jared Matury

Introduction

CatScript, a language that compiles to Java bytecode, operates at a high level similar to the C programming language. However, compared to C or Java, it lacks extensive refinement and functionality. The primary motivation behind developing the CatScript compiler was educational. The language encompasses essential features for practical functionality, including a static type system, basic control structures (such as IF statements, FOR loops, and FUNCTIONS), variable assignment, immutable lists, and boolean comparison operators. Additionally, it supports fundamental algebraic operations like addition, subtraction, multiplication, and division.

Type System

Type	Name	Java Mapping	Description
int	Integer	java.lang.Integer	32-Bit Integer
string	String	java.lang.String	String Value
bool	Boolean	java.lang.Boolean	Boolean Value
object	Object	java.lang.Object	Any Value
null	Null	java.lang.Object	Null Type
void	Void	java.lang.Object	Void Type

In CatScript, there are six fundamental data types: int, string, bool, object, null, and void. Notably, decimal value numbers and single-character values are excluded from these basic types, aligning them closely with their Java counterparts as outlined in the table above. When explicitly specifying the type of a variable, the syntax in CatScript resembles the following code snippet:

```
var variable: string = “Hello World!”
```

These variables can also be used to define new variables with type inference such as in the following:

```
var newVariable = variable + 1
```

The resulting value is "Hello World!1", where the integer value 1 is converted to a string and concatenated with the previous variable's value. The type of the resulting value is inferred accordingly. Further explanations of these operations will be provided in a subsequent section. However, trying to assign an object type value to an integer would lead to an error.

```
Var objectO: Object = 20
var intI: int = objectO
```

This is due to the incompatibility of those types and their inability to be cast from one to the other. On the other hand, all other variable types can be assigned to a variable of type object:

```
var stringA: string = “Howdy Partner!”
var objectA: object = stringA
var boolA: bool = false
objectA = boolA
var intA: int = 20
objectA = intA
```

Features

For loops and list literal:

For loops in CatScript iterate over collections and list literals are a collection of variables or expressions:

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

Output:

```
1\n2\n3\n4\n5\n
```

In this example, the loop iterates over each element in the array [1,2,3,4,5] and prints the individual elements. Loops are handy if you want to run the same code again with different values each time. Limitations for list literals are that the only way you can get the elements of the list is by using the for loop, we can’t do indexing. Another limitation of lists is they must contain the same data type. For the for loop there is one significant limitation, we can only loop through lists. We can’t loop on strings or through a range like in the Python programming language.

If Statement:

If statements in CatScript execute code given the condition is met:

```
if (x == 5) {
           print("x is 5")
} else {
           print("x is not 5")
}
```

Output for if, x is equal 5:

```
x is 5
```

Output for if, x is not equal to 5:

```
x is not 5
```

In this example, the if statement executes the print statement "x is 5" if the value of x is indeed 5. This is useful if you want to perform different actions for different decisions. The ‘and’ and ‘or’ usually found in programming languages are not supported in catscript.

Print Statement

Print statements in CatScript output certain expressions to the console or terminal:

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

Output:

```
Hello world!
```

The `print()` statement is useful to display information to the user, and it can help with debugging if you want to print variables you are using at different points in the program.

Function call, definition, and return

Function definitions in CatScript allow encapsulation of code, and it can't be executed unless it is called. The definition has parameters also known as variables that can be assigned to expressions. Function calls in CatScript allow functions to be called and run. In addition, function calls can assign values to the definition parameters. Return calls in CatScript return a value back to the caller of the function:

```
function calculate_sum(a,b){
    return a + b
}
var x =  calculate_sum(3,5)
print(x)
```

Output

```
8
```

In this example, the function calls calculate_sum(3,5) and assigns the a value to 3 and the b value to 5 allowing our definition to execute its encapsulated code. Return allows the var x to be assigned to 8. Functions are useful to store code and make sure outside variables don’t affect the outcome of that code. Returning is useful so we can get variables back to its caller.

Equality Expression

Equality Expression in CatScript is used to compare two different variables or expressions. The == represents equals and the != means not equals:

```
var x = 5
var y = 5
var z = 4
print(x == y)
print(x != y)
print(x == z)
print(x != z)
```

Output

```
true\nfalse\nfalse\ntrue\n
```

In this example we compare x to y and z the outcome is either true or false depending on whether the condition is true. The limitation of this is that we can't see if strings are equivalent to each other; it always returns false.

Comparison Expression

Comparison Expression in CatScript is used to compare two different variables or expressions. The <= represents less than or equal and the >= means less than or equal:

```
var x = 5
var y = 5
var z = 4
print(x <= y)
print(x >= y)
print(x < y)
print(x > y)
print(x >= z)
print(x <= z)
print(x > z)
print(x < z)
```

Output

```
true\ntrue\nfalse\nfalse\ntrue\nfalse\ntrue\nfalse\n
```

In this example we compare x to y and z the outcome is either true or false depending on whether the condition is true. Catscript does not support chaining. Chaining is comparing multiple comparison expressions at the same time.
The Equality and Comparison expressions are also known as Boolean expressions. These are useful to build logic and find answers.

Additive Expression

Additive Expression in CatScript is used to add or subtract two or more integers:

```
var a = 4
var b = 3
var ab = a + b
var aa = a - b
print(ab)
```

In this example, we assign var ab and aa to the additive expression.

Output

```
7\n1\n
```

Factor Expression

Factor Expression in CatScript is used to multiply or divide two or more integers:

```
var a = 4
var b = 2
var ab = a * b
var aa = a / b
print(ab)
print(aa)
```

Output:

```
8\n2\n
```

In this example, we assign var ab and aa to the Factor expression. The Factor and Additive expressions allow us to do mathematical calculations so we wouldn't have to do them by hand.

Unary Expression

Unary Expression in CatScript is used to get the inverse value of an expression or variable:

```
var a = 4
var b = 2
var ab = -a * b
var aa = not true
print(ab)
print(aa)
```

Output:

```
-8\nfalse\n
```

In this example, we assign var ab and aa to the Unary expression. This is useful because we can now work with negative numbers and help us satisfy conditions if something isn't met.