

Compilers: Capstone Portfolio

Alex Krings & Nic Ceccanti

CSCI 468: Carson Gross

May 5, 2023

CSCI 468

Section 1: Program

The goal of the capstone was to learn how to implement a compiler for a programming language called CatScript. Throughout this project I learned how to traverse the grammar of a programming language to scan tokens, parse the tokens using recursive descent, and then generate Java Bytecode from the parse tree. This project was developed using Java, and a zip file of the source code has been attached to the submission alongside this portfolio.

Section 2: Teamwork

The teamwork portion of this project was split into two groups: primary engineer (team member one) and the documentation/testing engineer (team member two). Team member one spent the majority of the time on the project and was tasked with building the compiler.

Team member two created intensive test cases and created a CatScript guide to satisfy the requirements for section 4. Below are the tests generated by team member two which are also located in the following directory `/src/test/java/edu.montana.csci.csci468/capstone/CapstoneTest.java`.

```
@Test
void voidFunctionWithForLoop() {
    assertEquals("1\n2\n3\n4\n5\n",
        executeProgram("function foo() {" +
            "for (x in [1,2,3,4,5]) {" +
            "print(x)" +
            "}}\n" +
            "foo()"
        ));
}

@Test
void voidForLoopWithPrintingComparsionExpression() {
    assertEquals("false\nfalse\nfalse\ntrue\ntrue\n",
        executeProgram("for( x in [1,2,3,4,5] ) {" +
            "print(x > 3)" +
            "}"
        ));
}

@Test
void voidPrintMaxFunction() {
    assertEquals("926\n",
        executeProgram("var max = 0\n" +
            "for (x in [926,2,93,432,564]) {\n" +
            "    if( max < x ) {\n" +
            "        max = x\n" +
            "    }\n" +
            "}"
        ));
}
```

```
        "print(max)\n"
    ));
}
```

The first test created ensured that the for loop iteration was correct and that the iterator value was storing the correct value.

The second test created help check that a `print` statement with a conditional was able to work with an iterator value.

The final test was another `for` loop test with an `if` statement on the inside.

All of these helped better test our software to make sure that it was running correctly.

Section 3: Design pattern

One design pattern that was used for this project was Memoization. This is located in `src/main/java/parser/CatScriptType` starting at line 39 and ending on line 47. For convenience, the code is also attached below.

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPE = new
HashMap<>();
    public static CatscriptType getListType(CatscriptType type) {
        CatscriptType listType = LIST_TYPE.get(type);
        if(listType != null){
            listType = new ListType(type);
            LIST_TYPE.put(type, listType);
        }
        return new ListType(type);
    }
}
```

What the above code does is store the initial type from when it was first called, so it doesn't have to reinitialize the type. This saves computational time, as you don't have to create and initialize a type every single time. Which is essentially just cacheing.

Section 4:

Catscript Guide

This document is a guide on how the Catscript language works. This is intended to satisfy section 4 of the Catscript portfolio.

Introduction

Catscript is a simple scripting language. Here is an example of a function that prints out all the values in a list:

```
function foo() {  
  for (x in [1,2,3,4,5]) {  
    print(x)  
  }  
}  
  
foo() // 1 2 3 4 5
```

Features

Comments

Catscript supports commenting in code. Catscript comments can be specified by using `//`. All text after the `//` will be included in the comment until the end of line.

```
// This is a comment  
print("hello world") // This is also a comment.
```

Catscript Types

CatScript is statically typed, with a small type system as follows:

- `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

Catscript has types similar to those of other languages. Integers are 32-bit and are specified using the `int` keyword. Strings are similar to that of a Java string and are specified using the `string` keyword. String values are specified by encapsulating the string value in double quotes. Booleans are true and false values, specified by the `bool` keyword. Boolean values in Catscript are specified by using `true` or `false` keywords. The `object` type is much like that of the Object type in Java. The `object` type can be specified by using the `object` keyword. Values of any type can be assigned to the `object` type. A variable of type `null` may not be explicitly declared as `var x : null = null` but may be assigned implicitly as such, `var x = null`. The only value that can be assigned to a variable of type `null` is `null`, no other variable types can be assigned to a variable of type `null`. Additionally, the `null` value be implicitly changed the string value of `"null"` when printed or concatenated.

Catscript variables are statically typed, meaning after initialization values of a different type from a variable cannot be assigned to it. There are two exceptions to this rule however, the `object` type can be assigned the value of any type. Secondly, the value `null` may be assigned to a variable of any type.

```
var x : int = 0
var y : bool = true
var z : object = false
var a = null
var b : string = "hello world"

x = null
```

Catscript Lists

One of the unique types in Catscript are lists. A list can be of the type of any other type. Lists in Catscript are immutable. Once declared, they cannot be modified. However, lists of the same type are covariant. A list can be assigned to another list, if the type of list being assigned is assignable to the type of the list being assigned to. Nested lists are supported and there are no limits on the dimensionality of list, as long as values in the lists follow the assignability rules of the type of the list. Lists can have a type explicitly assigned by using `:` followed by the keyword `list`. Next, after `list`, the specified type of list surrounded by `<>` such as `list<string>`. A list of type `object` is able to hold values of all different types at the same time.

```
x : list<int> = [1,2,3]
y : list<object> = [[1, "foo"],[null, true]]
z = [1,[2,3]]
```

Print Statement

The print statement in Catscript, will print out a specified value to the terminal. The print statement is specified by using the `print` keyword, followed by parentheses where the desired expression to be printed is contained. Any valid expression can be put inside parentheses of the print statement. The evaluated value of the expression within the parentheses will print to the terminal.

```
print("Hello World") // Prints hello world
```

Variable and Assignment Statement

The variable statement in Catscript assigns some value to a name. This is done by specifying the `var` keyword, then the name of the variable. Optionally, you can explicitly specify the type of the variable by placing a `:` then the Catscript type expression. This will enforce the type of the variable. If the type is not specified, the Catscript Parser will automatically assign a variable type to it implicitly. Next specify a single equal sign then the expression or value of the desired value to be assigned to the variable.

```
var string = "Hello world"
var num : int = 1
```

Comparison Expression

Catscript comparison expressions compare two integer values that evaluate to a boolean value based on the outcome of the expression. Catscript uses the common symbols for comparisons. `>=` greater or equal, `>` greater, `<=` less than or equal, and `<` less than.

```
var x = 1 > 1
var y = 1 >= 1
var z = 1 <= 1
var a = 1 < 1
```

Equality Expression

Catscript equality expressions evaluate whether two values of any type are equal or not equal. Catscript supports the commonly used comparison equals `==` and comparison not equal `!=` symbols. Equality expressions evaluate to a boolean value.

```
if(x == 1) {
  print("foo") // If x equals 1 print foo
}

if(x != 1) {
  print("fee") // If x does not equal 1 print fee
}
```

Additive and Factor Expressions

Catscript supports the basic addition, subtraction, multiplication and division expressions.

For divisional operations, the resulting value will only evaluate to an integer, any decimal place or remainder will be dropped from the value. For example `print(3 / 2)` will print out the value `1`. Catscript does not support floating point values.

```
var x = 1 + 1
var y = 1 - 1
var z = 1 * 1
var a = 1 / 1
```

Concatenation

Catscript supports the concatenation of `string`, `int` and `null` types together to produce a string, much like that of the Java language. By using the `+` operator between values or variables, their string values will be concatenated. Concatenation is implicitly specified by checking if one of the values in the expression is of a `string` type. `int` types will take the string form of its value. For example `1` becomes `"1"`. The null value can also be used in concatenation, the `null` value will convert to the string value of `"null"`.

```
print("a" + 1) // "a1"  
print(null + "foo") // "nullfoo"
```

Unary Expressions

Catscript supports two unary expressions. The **not** keyword is the equivalent to the logical NOT operator. The **not** keyword flips the Boolean value given in the expression. **not** can only be used with Boolean values. Secondly, the **-** can be used as a unary operator to specify a negative integer value. The **-** operator can be put in front of an **int** value to make an **int** value negative.

```
if(not x) {  
  print("foo") // If x is false print  
}  
  
print(-1) // Prints -1
```

Parentheses

Catscript supports the surrounding of any expression in parentheses. One might surround an expression in parentheses to give precedence to a specific part of an expression so that it is executed first in its evaluation. The more nested an expression is in parentheses, the higher its priority in the evaluation,

```
print((1 + 2) * 3) // Prints 9
```

For Loop

The for loop in Catscript only supports iterating over lists. The for loop is initiated by using the **for** keyword followed by a set of parentheses. Inside the parentheses you specify the iterator variable and the list to be iterated over. The iterator variable and the list to be iterated over is separated by the keyword **in**. The iterator variable, in the example **x**, takes the value from the list that the iterator is currently on. For each iteration, the code within the curly brackets will be executed until the iterator reaches the end of the list. The iterator variable is only visible in the scope of the for loop body. For loops can only iterate over lists, iteration given a conditional statement or counter is not supported.

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

If Statement

To initiate an if statement in Catscript use the **if** keyword, followed by a conditional expression encapsulated by parentheses. Next, within a set of curly braces, specify code instructions to be executed if

the conditional statement evaluates to true. Next, optionally you can specify an else statement by using the keyword **else**, followed by code specified within a set of curly braces. The code following the else statement will execute if the conditional expression evaluates to false. If no **else** is specified and the conditional expression is false, code specified in the curly braces will be skipped and execution of code will continue after the if statement body. Catscript if statements only support if and else conventions, else if or elif style statements are not supported. However, any number of nested statements, including the if-else statements are supported within if and else statement bodies.

```
if ( x > 2 ) {
    print("foo") // If x is greater than 2 print foo
} else {
    print("fee") // If x is less than or equal to 2 print fee
}

if(x == -1) {
    print("faa") // if x equals -1 print "faa"
}
```

Function Calls and Definitions

Catscript functions allow for defining and invoking of subprograms. To define a function, first use the **function** key word followed by a function name. Follow this with a set of parentheses, within these parentheses specify the arguments of the subprogram. Each of the arguments will be specified by a variable name followed by an optional type expression. The specified type and its associated identifier will be separated by a **:**, similar to that of the variable statement. Each of the arguments will be separated by a **,**. Functions can have none to many arguments. A return type is specified by a **:** after the set of parentheses surrounding the arguments, then the type expression. To return a value from a function a return type must be specified. If the function is void, or does not return any value, do not specify a return type. If the function has a return type specified the function must have a return statement. Additionally, if there are possible changes in control flow of a function, every single branch of the control flow must end on a return statement. For example if there is an if-else statement, and there is a return statement in the if statement body, there must also be a return statement in the else body or another return statement after the if-else statement. A return statement can be specified by using the **return** keyword followed by some expression, but only if a return type is specified. Void function do not need to use return statements. In a void function, a function with no specified return type, can optionally be terminated by using the return statement with no associated value or expression. Additionally, if the function is void then return statements are not required in all branches of the control flow. Return statements in void functions can be used in some branches of the control and not others. There is no requirement for all branches have to have return statements in all possible branches in void functions.

The function call invokes the code in the function definition. The function call is the name of the function followed by a set of parentheses. Any input arguments needed for the functions are specified in the same sequential order as the function definition, separated by **,** within the parentheses. You must have an input parameter for each argument specified in the function definition and correct variable type if it is specified.


```
function foo() {
    print("hello world")
}

foo() // Prints hello world

function helloWorld(x, y) : string { // Concatenates the variables x and y
and returns the value
    return x + y
}

function add(a, b) : int {
    if (a < 0) {
        return 0
    } else {
        return a + b
    }
}

print(add(-1, 5)) // Prints 0

var x = helloWorld("hello ", s)
print(x) // Prints hello world

// This function will not do anything
function foo() {
    var x = 3

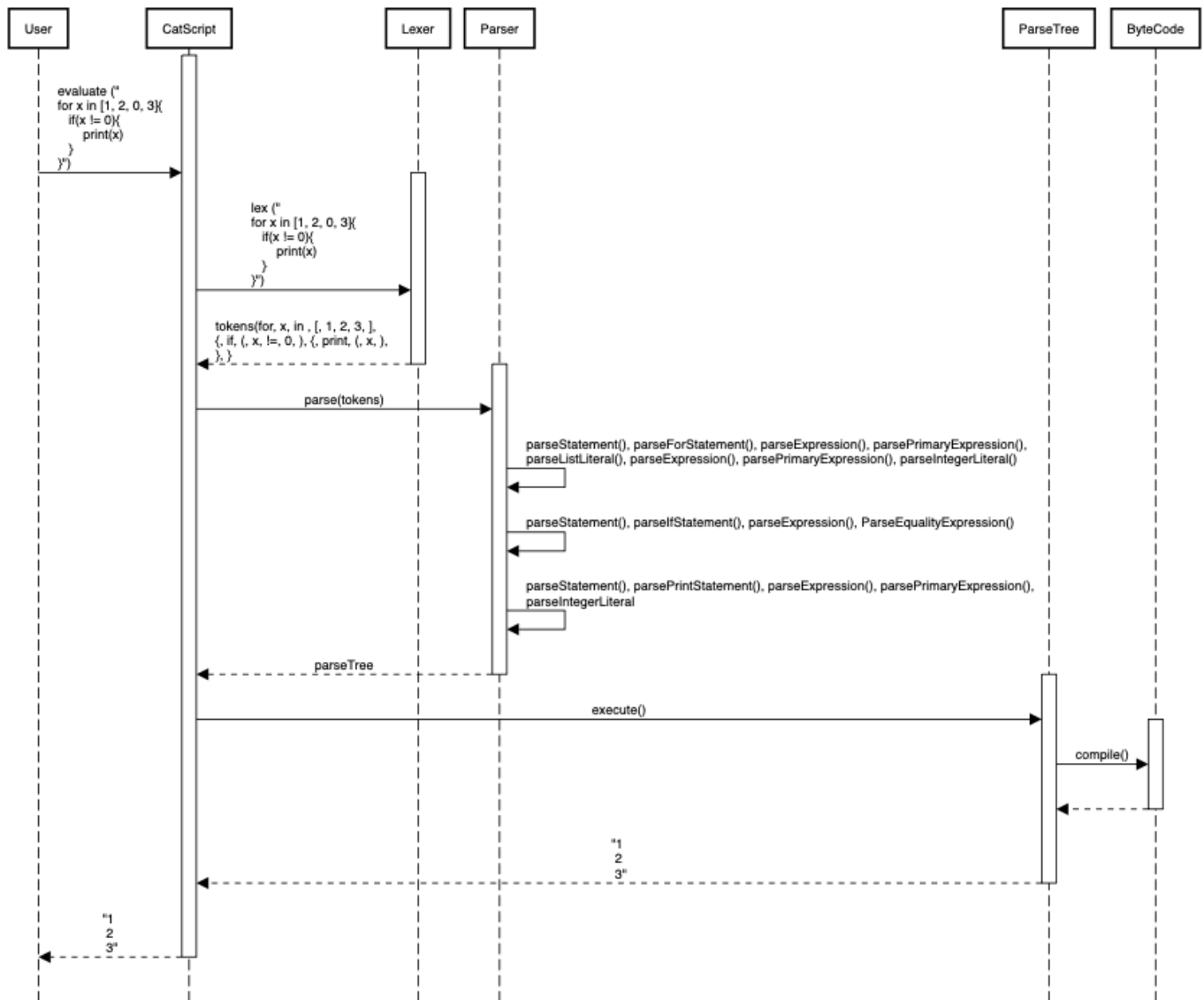
    if(x == 2) {
        return
    }
    print("foo")
}
```

Section 5: UML

Below is a sequence diagram of a high-level overview of a compiler.

I would like to note that in order to conserve some vertical space, I grouped the parser elements into the main groupings. In reality, they would all be on their own.

Catscript For Loop Sequence Diagram



Section 6: Design trade-offs

The biggest design trade off for this project was that we created a parser by hand, rather than using a parser generator. There are numerous pros for doing and creating the parser by hand. The biggest one being I was able to learn more about compilers and parsers by doing it this way. Another pro for creating the parser by hand is being able to debug it. After looking at what an auto-generated parser and tokenizer looked like our tokenizer was far smaller and easier to read. The auto generated parser was smaller, but still unreadable.

The cons of handwriting the parser is that it takes a lot longer. The auto generated one creates it very fast and accurate. And if you like the **Visitor Pattern**, then the auto generated ones are more geared for that.

Section 7: Software development life cycle model

The life cycle model that was used to complete this project was Test Driven Development (TDD). This model is where there are software requirements which are then used to create tests before the project has been fully developed. At the start of the project, there was a test suite given. Within that suite there were sub-folders of tests that accompanied aspects of a compiler, such as tests for the `bytecode`, `eval`, `parser`, and the `tokenizer`.

TDD helped our project immensely for a couple of different reasons. The first one being, it gave us a great cornerstone to kick off from and start building our compiler. Another reason TDD was helpful was we could see what our final result was supposed to look like. That helped with debugging and seeing what we were doing wrong.