CSCI468 Compilers Capstone

Author: Emily Ingbertsen
Partner: Carlee Joppa

May 2024

1 Program

A zip file titled source.zip in this same directory contains all the code pertaining to this program. This program pertained to writing a compiler for own coding language CatScript. As such, this program required tokenizing, parsing, evaluation and bytecode creation from written CatScript code.

2 Teamwork

For this capstone project, my primary contribution was writing all the code for the program and ensuring tests written for this program passed. My contributions in code were writing the tokenizer, parser, evaluations for expressions and statements, and bytecode. My team member was responsible for writing the documentation on the CatScript programming language (provided in Section 4) as well as writing a series of tests to ensure the final product properly functioned. I spent roughly 95% of the time on this project as I was responsible for writing all the code compared. My partner's 5% of the time contributing to this project through writing documentation and tests.

The tests they provided are as shown below:

```
public class PartnerTests extends CatscriptTestBase {
@Test
public void addResultOfFunctions() {
    Object program =
    executeProgram("function x(a : int, b : int, c : int) : int { " +
            "return a + b + c } " +
            "function y(a : int, b : int) : int { return a * b } " +
            "var z : int = x(1, 2, 3) + y(2, 3)" +
            "print(z)");

    assertEquals("12\n" , program.toString());
}

@Test
void assignmentStatementWorksInForLoopProperly() {
    assertEquals("6\n9\n12\n", compile("var y = 3\n" +
            "for( x in [1, 2, 3] ) {\n" + "  y = y + 3" +
            "  print(y)\n" +"}\n"));
    assertEquals("200\n100\n50\n", compile("var y = 400\n" +
            "for( x in [1, 2, 3] ) {\n" + "  y = y / 2" +
            "  print(y)\n" + "}\n"));
    assertEquals("4\n8\n24\n", compile("var y = 4\n" +
            "for( x in [1, 2, 3] ) {\n" + "  y = y * x" +
            "  print(y)\n" + "}\n"));
}
```

```
    @Test
    public void parseExpressionWithParens() {
        Statement expr = parseStatement("print(((9 + 3) / 2) / ((10 / 2) - 3))");
        assertFalse(expr.hasErrors());
        Statement wrong = parseStatement("print(((9 + 3))", false);
        assertNotNull(wrong);
        assertTrue(wrong.hasErrors());
    }
}
```

3 Design Pattern

The main design pattern used for this project was the Memoization pattern. The memoization design pattern is generally used to speed up computationally expensive programs by caching values when they are first computed and returning the cached result if requested again. However, for this program I mainly use this pattern to reduce the amount of repeated objects I am creating by caching them.

This pattern was used in this program in the CatscriptType class's getListType function which is used to return a list type given the specified type of the list. In the case of this program, I use a static hashmap to cache the already created listType without needing to recreate the same object repeatedly. As such, if we have not created an object of that listType yet, then a new object of that listType will be created and then cached to the hashmap. Otherwise, if we have already created an object of that listType, then it can be retrieved from the hashmap.

By using the memoization pattern in this way, we greatly reduce the number of repeated objects we are creating and still maintaining quick and efficient code.

The memoization pattern is shown in the code below:

```
private static HashMap<CatscriptType,ListType> listTypes = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {

    if(listTypes.get(type) != null){
        return listTypes.get(type);
    } else {
        ListType listType = new ListType(type);
        listTypes.put(type,listType);
        return listType;
    }
}
```

4 Technical Writing

The following is the documentation written to describe the CatScript programming language and its features:

Introduction

CatScript is a simple statically-typed scripting language. Syntax-wise, CatScript is heavily influenced by C style syntax. CatScript is a functional programming language and as such does not support classes.

This is a guide to the CatScript programming language, and as such will define all features of the language.

Here is a simple example of CatScript:

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

Features

CatScript includes a variety of features including for loops, functions, and if statements.

Type Literals

The CatScript type system supports int, string, bool, null, object, and list.

Integer Literal

The integer literal can support any 32-bit integer. This means that fractions and decimals are not supported in Catscript. The type of integer is shorted to ‘int‘ for use in this language.

Here are some examples of values that are supported:

```
 4
-8
993491
```

String Literal

CatScript also supports Java-style strings. Strings must be in quotation marks (“”“). The type name of string literals in CatScript is ‘string‘

Here are some examples of strings in CatScript:

```
"Hello World"
"true"
"16"
```

Boolean Literal

Boolean values are always represented by opposite values, but unlike other types, booleans only have two values. In CatScript, the two values are ‘true‘ and ‘false‘, also shown below. The type name of boolean literals in CatScript is shortened to ‘bool‘.

Here are some examples of boolean values in CatScript:

```
true
false
```

Null Literal

Like in other programming languages, null is a type that be assigned to anything. This literal also has no value and often takes the role of a placeholder.

```
null
var x = null
```

Object Literal

The object literal can be assigned any type of value. While CatScript does type inference, there are some instances where elements are assigned type ‘object‘ automatically, such in the case that parameters don't have a specified type in the function declaration.

Here are some examples of valid values for objects:

```
"Hello World"
true
16
```

List Literal

Components of lists can be of any type, including another list. Lists that are made up of multiple component types are automatically lists of objects.

Here are some valid CatScript example lists:

```
var a = ["hello", "world"]
var b : list = [1, false, [4, 2]]
var c : list<bool> = [true, false]
```

Variable statement

Variables can be assigned by using the keyword var, followed by a variable name. CatScript does type inference, so types can be excluded or included, as seen in the examples below.

The example list ‘z‘ shown below will have a component type of object.

```
var x = "Hello World"
var y = 3
var z = [1, true, "hello"]
```

To include a type, use a colon and the type in between the variable name and ‘=‘. Lists with an explicit type of list can also choose to specify the explicit type of the component. Lists that specify the component type must do so with ‘list¡type¿‘, with any Catscript type, after the colon.

```
var a : bool = false
var b : list = [1, false]
var c : list<bool> = [true, false]
```

Operators

Unary Operators

In Catscript there are two unary operators ‘-‘ and ‘not‘. The operator ‘-‘ can be used to negate ints, while ‘not‘ can be used to negate booleans.

```
var x = -19
var z = not true
```

Equality Operators

To determine whether two elements are equal, the ‘==‘ operator can be used, which will return true if they are in fact equal. The ‘!=‘ operator can be use find whether two elements are not equal. The equality operators can be used to find equality of all types.

```
var x = 19
if (x != 19){
    print(x)
}
else if(x == 19){
    x = 5
}
```

Comparison Operators

The operators for comparison are as follows: ‘¿‘ (greater than), ‘¡‘ (less than), ‘¿=‘ (greater than or equal to), ‘¡=‘ (less than or equal to). Unlike some of the other operators, comparison operators can only be used with int types.

```
var x = 19
if (x > 19){
    print(x)
}
else if(x <= 19){
    x = 5
}
```

For Loop

For loops can be used to iterate over a list.

For loops must start with the keyword ‘for‘ followed by parens containing a python style ”variable ‘in‘ list” syntax shown below. The iteration variable used inside the for loop (‘x‘ in the example below) will be first be assigned the value of the first element in the list. Once the body statements of the for loop are executed, the process will repeat with the second value in the list assigned to the iteration variable and so on through the list.

Unlike some languages (Java and others) curly brackets must be used to indicate the beginning and end of the for loop, like is shown in the example below.

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

If Statement

If statements can be used to choose whether to execute certain statements based on some condition.

In CatScript, if statements can optionally include else if and else conditions, as shown below. Else conditions are executed only executed when the if and else if statements are false. For if and else if, the conditional must be in parentheses after ‘if‘ and before the curly brackets.

Like for loops, if statements also must include curly brackets to start and end the body statements.

```
var x = 10
if(x > 10) {
    print(x)
}
else if(x == 10) {
    x = 2
}
else {
    x = 7
}
```

Additive expressions

In CatScript, addition and subtraction can be done using the usual operators (‘+‘ and ‘-‘ respectively), and the ‘+‘ operator can also be used to do string concatenation. For subtraction, the only int types can be used, and for addition, only string and int types can be used. If a string and int are added, the result will be a string (the int will automatically be cast to a string).

```
var x = 19 - 2
var y = x + 12;

print("x is " + x)
```

Factor expressions

Multiplication in CatScript can be done using the ‘*‘ operator, and division can be done using the ‘/‘ operator.
In CatScript, multiplication and division can only be done with int types.

```
var x = 10 * 2
var y = x / 4
```

Print Statement

Print statements in CatScript start with the keyword ‘print‘ followed by parentheses. Whatever is inside the parentheses is outputted to the console. Concatenation can be done inside the print statement using the ‘+‘ operator.

```
print("Hello World!")
```

Assignment Statement

Once variables are declared, they can be assigned to a different value; however this value must be a type that is assignable. This means that if a variable was initially declared as a string, it cannot be assigned ‘false‘, which is a bool type, later.

Because lists are immutable in CatScript, lists elements cannot be changed using assignment statements.

```
var x = 8
x = 10

var y = "hello"
y = "goodbye"
```

Function Declaration

Functions in Catscript must start with the keyword ‘function‘, followed by the name of the function. Any parameters to the function must be specified inside the parens. The type of parameters can be added or omitted as shown below in the second example. If omitted, the type of the parameter will automatically be assigned as object. Like the for loop and if statement, curly braces must be used to denote the start and end of the body statements of the function.

```
function cost(a : int, b : int) : int {
    return a + b
}

function printB(a : int, b) : int {
    print(b)
    return a
}
```

Function Call Statement

In order to call functions in CatScript, the function name used in the function definition must be used. This should be followed by parentheses containing parameters to the function separated by commas. If a function does not take any parameters, the function name should be followed by empty parentheses. When functions have a return value, it will be returned to where the function was called, like in the example below, where the return value of the cost function will be assigned to ‘x‘.

```
var x = cost(3, 4)
```

Return Statement

The return type can be specified following the parens with a colon and the type. If no return type is specified, the automatic return type will be void. In this case either an empty return statement can be used (shown in the first function), or the return statement can be omitted completely. Return statements can only be used inside of functions, and will produce an error anywhere else.

```
function printA(a : int) {
    print(a)
    return
}
```

If a return type is specified in the function declaration, a value with that type must be returned. This can be done with a return statement as the last line of the function, or if different return statements based on a condition are desired, return statements may be used in if statements. This option requires that all conditions have a return statement, shown below.

```
function evaluate(a : int) : int {
    if(a > 5){
      print(a)
      return a
    } else {
        return a + 2
    }
}
```

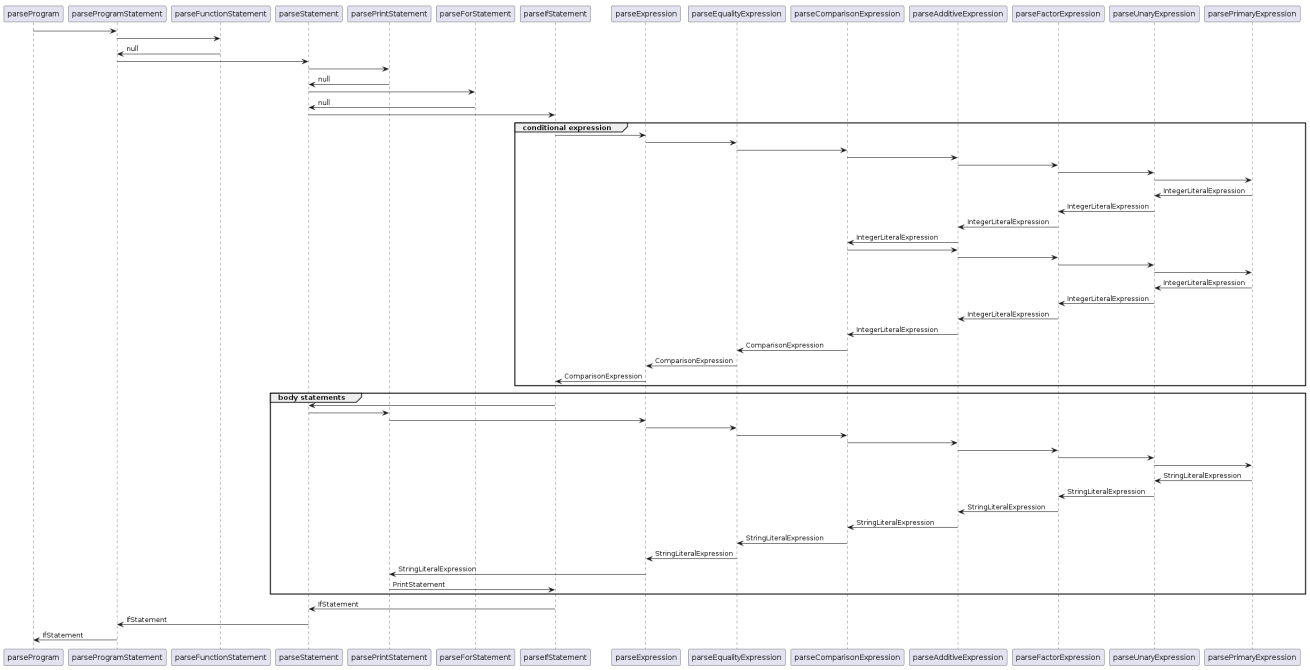
5 UML

The UML diagram found in this same directory represents the process of parsing the following snipet of CatScript code:

```
if( 1 < 5 ){ print("Hello World") }
```

The diagram highlights the aspect of recursive descent present in the implementation of the parser as we can visualize the recursive calls to other functions within the parser until the correct function parses the code.

The parsing begins by parsing a statement from which it calls the function parseIfStatement. From there, the function parses the conditional expression which follows recursive descent by calling the parseExpression function until it reaches the parsePrimaryExpression function. It then recursively goes back up the call stack where it parses the less than symbol and calls parseComparison expression. It then proceeds to once again recursively call down to parsePrimaryExpression for the second integer of the comparison expression. Once the comparison expression is parsed, it is returned to the if statement object where it then proceeds to parse the body statements by calling parseStatement. This function calls down to parsePrintStatement which then calls parseExpression to recursively call down to parsePrimaryExpression which returns the string literal object. Once the printStatement is done parsing, it returns back to the if statement object.From there, the if statement object is recursively returned back up the call chain.



6 Design Trade-Offs

In this project, I decided to use a recursive descent parser rather than using a parser generator for the design of the parser.A parser generator generates code automatically given grammar specified from the programmer.A recursive descent parser is handwritten by the programmer and focuses on using recursion to recursively parse tokens while also backtracking when tokens do not match until there are none left and the program has been completely parsed. However, instead of a parser generator, I decided to use a recursive descent parser instead for my design.

The trade-off here is that a parser generator is less handwritten code compared to the recursive descent parser which is completely written by hand. Additionally, there is much less infrastructure to verify the integrity of in the parser generator in comparison. However, I prioritized the control recursive descent parsers allow the programmer as well as the readability as parser generators often require terse code and obscure syntax. Additionally, parser generators can be incredibly difficult to debug and recursive descent parsers allow for a clean infrastructure to clearly follow along with while debugging.

7 Software Development Life Cycle Model

The model I used to develop this project was Test Driven Development. Test Driven Development is an approach in which the developer write tests for the code before writing the code. These tests are meant to reflect the desired behavior expected from the code after being written. Following the writing of the tests comes the writing of the code to meet the desired behavior the test expects. Once complete, the test can be run to see if the code meets the requirements of the test. If not, then the developer must return to the writing stage to ensure the test passes. For our project, the tests were both written by the instructor as well as my partner.

The benefit of using Test Driven Development in this project is that it provides an immediate feedback loop during the coding phase and allows for faster awareness on the correctness of the code written. Additionally, it allows the developers to consider and write out the requirements of the code before writing it.