CSCI 468, Compilers

Spring Semester 2024

Burgin Luker, Bennet Sampson

Section 1:

Visit the zip file that has been provided to see the source code.

Section 2:

Regarding teamwork, my team member, referred to as team member 1, wrote me a series of tests that I could use to test my compiler (See figure 1). These tests included testing catscript features such as lists with strings, ensuring if statements have a condition to test, garuanteeing that additive expressions can be passed as function arguments, and that list literals can be passed as function arguments.

Figure 1.

```
@Test
// This test ensures that the List Literal expressions work with String Expressions in them
public void parseStringListLiteralExpression(){
    ListLiteralExpression expr = parseExpression("[\"strings\", \"in\", \"list\"]");
    assertEquals( 3, expr.getValues().size());
    StringLiteralExpression firstStr = (StringLiteralExpression) expr.getValues().get(0);
    assertEquals("strings", firstStr.getValue());
}

@Test
// This test ensures that the If Statement requires a condition to test and throws an error otherwise
public void ifStatementEnsuresCondition() {
    IfStatement expr = parseStatement("if(){ print(x)", false);
    assertNotNull(expr);
    assertTrue(expr.hasErrors());
}

@Test
// This test ensures that an Additive Expression can be passed as an argument for a Function Call Expression
public void parseFuncCallWithAdditiveExpressionWorks() {
    FunctionCallExpression expr = parseExpression("foo(1 + 2)", false);
    assertEquals("foo", expr.getName());
    assertEquals(1, expr.getArguments().size());
}



@Test
```

```
// This test ensures that a List Literal Expression can be passed as an argument for a Function Call Expression
public void parseFuncCallWithListLiteralExpressionWorks() {
    FunctionCallExpression expr = parseExpression("foo([1, 2, 3])", false);
    assertEquals("foo", expr.getName());
    assertEquals(1, expr.getArguments().size());
}
```

Thankfully, no bugs were uncovered in this testing, which we believe was due to the rigorous testing that was done during the development of the software. That said, the tests provided by team member 1 allowed me to stress my compiler in ways that I had not thought of before, forcing me to understand the mechanics of my compiler even deeper to verify both that the parser can handle the test cases and why.

Additionally, I wrote team member 1 a series of tests that I also tested on my own compiler. This allowed us both to test our products and different ways and discuss where our projects were different and similar, which resulted in a greater learning experience for both of us.

Section 3:

For my project, I utilized the design pattern of memoization. In java, when a new instantiation of a class is needed, you use the new keyword to create memory for the new object. This process can be memory intensive if you have many objects created and can consume memory at overly costly rates. However, this issue can be solved with memoization.

Memoization, which involves caching the result of an expensive function call and returning that value if it exists, solves this issue. This issue arose regarding lists of certain types in catscript. Instead of making a new type of list every time one was needed, through memoization I was able to check my cache if that list exists, then if it did return that object rather than having to make a new one every time (Figure 2).

Figure 2.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {

    if(cache.containsKey(type)){

        return cache.get(type);

    }

    else{

        ListType listType = new ListType(type);

        cache.put(type,listType);

        return listType;

    }
```

Section 4:

Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

Catscript is a simple scripting language. Here is an example:

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

Features

Here are the main features of the CatScript Programming Language, split into their two main categories: Expressions and Statements

Expressions

String Literal Expressions

String Literal Expressions are expressions that are used in CatScript to create words or sentences using a collection of characters. Strings are denoted by using a set of quotation marks around the string.

Here is an example:

```
var myString = "hello"
var sentence = "This is a sentence of strings"
```

Integer Literal Expressions

Integer Literal Expressions are a simple expression for defining integers in the CatScript programing language. Below is an example of this:

```
var z = 1
```

Boolean Literal Expressions

Boolean Literal Expressions are very simple expressions in CatScript that allow a user to use the Boolean values true and false.

Here is an example:

```
true
false
var a = true
```

Null Literal Expressions

The Null Literal Expression in CatScript is used to express the absence of a value of an expression. Here is an example:

```
var x = null
```

List Literal Expressions

List Literal Expressions are CatScripts implementation of an ordered collection of data. Lists can contain any type of data in CatScript as long as all values are of the same type. List Literal Expressions can be implicitly or explicitly typed.

Here is an example:

```
[1, 2, 3]
[true, false, true, false]
var strList : list<string>  = [list, of, strings]
```

Type Literal Expressions

Type Literal Expressions are the expressions that we use in CatScript to denote the types of our expressions and statements. In CatScript we have types int, string, bool, object.

Here is an example of using these types:

```
var x : int = 1
var y : string = "String"
var z : bool = true
```

Equality Expressions

Equality expression are expressions in the CatScript programming language that allow users to measure equality of expressions. Users can measure the equality of both Integer Literal Expressions and String Literal Expressions and Boolean Literal Expressions.

Here is an example:

```
var x = 1
var y = 2
var z = 2
x != y


z == y
```

Comparison Expressions

Comparison expressions are expressions in the CatSrcipt programming language that allow users to compare values of expressions. Users can compare Integer Literal Expressions.

Here is an example:

```
1> 2 1< 2
2 >= 2 5 <= 10
```

Additive Expressions

Additive expression are expressions in the CatSrcipt programming language that allow users to add expressions together. Users can add Integer Literal Expression as well as String Literal Expressions and Variables.

Here is an example:

```
var x = 1
var y = 2
var z = x + y
var con = "con"
var cat = "cat"
var con-cat = con+cat
```

Factor Expressions

Factor expression are expressions in the CatSrcipt programming language that allow users to multiply and divide expressions. Users can multiply and divide Integer Literal Expressions.

Here is an example:

```
var x = 1
var y = 2
var z = x * y
var a = y / z
```

Unary Expressions

Unary expressions are used to negate another expression. Expressions that can be negated with a Unary Expression are any primary expressions, and another unary expression.

Here is an example:

```
-1
not true
not not true
```

Function Call Expressions

Function Call Expressions are used in CatScript to call a function that has already been defined within the users program. Function Call Expressions expect the name of the function followed by parentheses with an optional arguments list.

Here is an example:

```
foo()
bar(a, b, c)
y = fooBar()
```

Statements

For Loops

For loops are statements in CatScript that allow a user to iterate over a list literal expression in a CatScript program. For loops in CatScript do not require the user to specify the type of the iterator in the loop statement.

Here is an example:

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


}
```

If Statement

If Statements in CatScript allow a user to perform an action based on a condition they set. If the desired conditions are met, the body of the if statement will be executed. If statements can be used in the form of a single "if", or an "if-else" form.

Here is an example:

```
if (x > 10) {
    print(x)


}

if (x >= 10){
    print(x)
} else {
    print("Less than 10")


}
```

Print Statement

Print Statements in CatScript allow a user to print output to the console for the user to see. Print statements take in an argument, and the argument is what the user wishes to print or display.

Here is an example:

```
print("This is a print statement")
print(x)
```

Variable Statement

Variable Statements in CatScript are used for declaring variables. Variables can be declared with implicit typing or explicit typing. Variables must be given a value upon declaration and are initiated with the "var" keyword.

Here is an example:

```
var x = 10
var y : bool = true
```

Assignment Statement

Assignment Statements in CatScript are used for assigning values to variables. Values being assigned to the variable must be of the same type as the previously declared variable, and all variables can be assigned to null, regardless of type

Here is an example:

```
y = null x = 11
```

Function Definition Statement

Function Definition Statements in CatScript are used to define new functions in a CatScript program. Functions in CatScript can have a return type of any valid CatScript type, or a function can have a void return type and not return a value. Functions are declared by first using the "function" keyword followed by the name of the function, as well as the argument list. Arguments of a function are not required, and may be implicitly or explicitly typed.

Here is an example:

```
function x(a, b, c) {}
function x(a : object, b : int, c : bool) {}
```

Function Call Statement

Function Call Statements in CatScript are used to call and execute existing functions in a CatScript program. Here is an example:

```
x(1, 2, 3)
```

Return Statement

Return Statements in CatScript are used to terminate a function and return a value from the function. Return statements must be the same type as the function that it is returning from. Return Statements are triggered with the keyword "return" anf are followed by an expression.

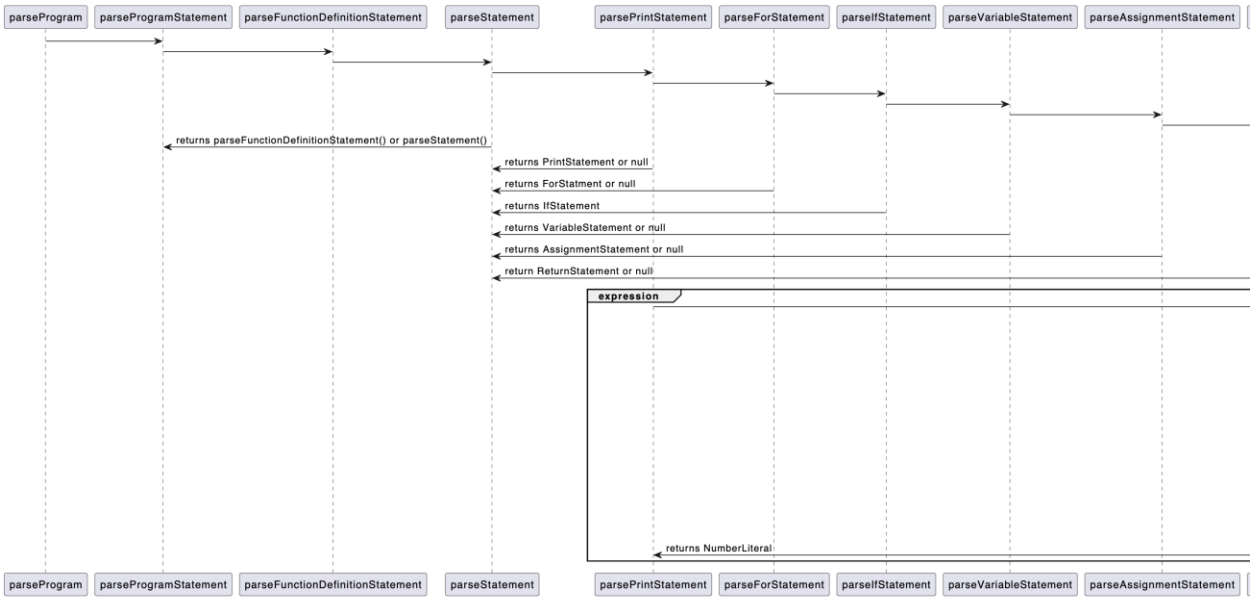
Here is an example:

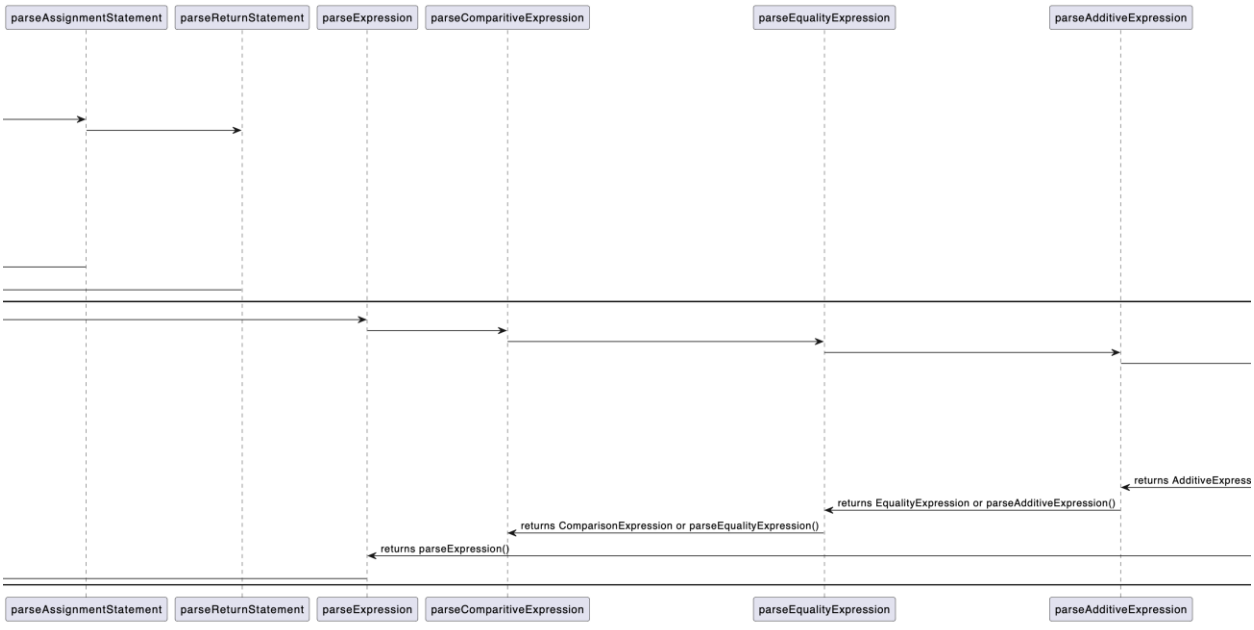
```
return true
function x() : int {return 10}
```

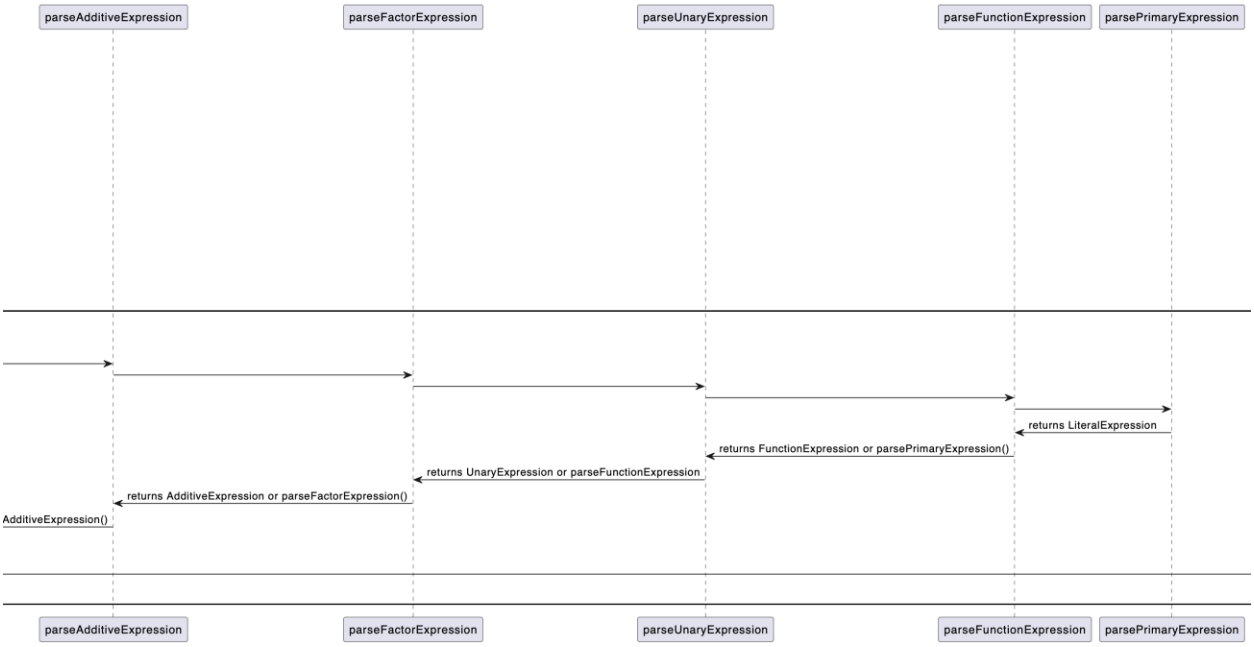
The above documentation for the catscript programming language was done by my team member team member 1. For this project we both created documentation for one another and then compared our documents to make sure we both covered every aspect of the scripting language catscript. We discovered that we both had very good documentation, the only difference was the order in which we decided to list things, but this is merely personal preference.

Section 5:

Below is the sequence diagram for parsing the statement print(2024) in catscript.







The above images represent the recursive decent algorithm implemented for parsing with a UML sequence diagram (Since the diagram was so long it was split into three images). To briefly describe what is seen in the diagram, first, each type of statement is attempted to be parsed. If that statement is found it will return its statement type to the function that called it. However, if the statement is not found, it will return a null value to the caller function and then a new statement will be attempted to be parse. Regarding expressions, since in the diagram we are parsing a print expression with a number literal, each type of catscript expression is attempted to be parsed and if that expression is not found the next type of expression is attempted. In this diagram a number literal is returned to a print statement.

Section 6:

For this project, it was a major design decision to use a recursive decent algorithm, implemented personally, rather than to use a parser generator. This decision was made for a variety of reasons, the main ones include the wide use of recursive decent in industry parsers and to facilitate greater learning when developing the parser.

Regarding industry use, since most industry parsers are done using recursive decent, I believe it is important to learn the algorithm. While parser generators are interesting and make for great thought experiments and can be used, the decision to use this project to prepare oneself for industry was prioritized over the experimentation with a parser generator.

With respect to facilitating greater learning, by writing the parsing logic I was forced to understand the recursive nature of the grammar and to thoroughly think through what was happening and what each step should be. This gave me a greater understanding of how grammar for programming languages is constructed and used. Additionally, it was felt like using a parser generator would have resulted in having to deal with learning obscure syntax and having less of an understanding of what was happening in the parsing logic, so overall less beneficial than learning recursive decent.

Section 7:

For this project test driven development was used. This involved generating files that tested four different functionalities of the compiler, tokenization, parsing, evaluation, and bytecode generation.

To test tokenization, I used a series of tests that involved all the token types in catscript. For example, to test the integer token and the plus token I used the string “1+2” with the assertion that this would produce the tokens INTEGER, PLUS, INTEGER, EOF (end of file). Moreover, for more advanced items such as lists I tested the string “[1,2,3]” with the assertion that the tokens produced would be LEFT_BRACKET, INTEGER, COMMA, INTEGER, COMMA, INTEGER, RIGHT_BRACE, EOF. I then repeated this strategy for all token types in catscript.

Regarding parsing, I used a of series of tests that involved all of the statements in catscript. For example, the test “if (x > 10) {print(x)}” with the assertions that an if statement and a print statement are generated and that there were not errors, I was able to determine if our parser was working correctly. By generating tests with the same logic for each statement allowed in catscript I was able to assess the functionality of my parser.

Evalution testing involved using a series of tests written to ensure that each expression and statement in catscript evaluates to a proper value. For instance, consider the following test:

```
void varInsideFunctionWorksProperly() {
    assertEquals("42\n", compile("function foo() : int {\n" +
            "  var x = 42\n" +
            "  return x\n" +
            "}\n" +
```

```
        "print( foo() )\n"));

}
```

With the assertion that 42 was produced I was able to test that function, types, variable statements, and return statements were all working correctly. Following this same logic for every statement and expression, with a corresponding assertion, I was able to determine all my statements and expressions were able to evaluate correctly.

Finally, bytecode testing was done with tests like the ones shown above, with the assertions being the same, except for that the value that was asserted was based on the bytecode generated by the compile method of my compiler. If the assertions passed, this meant that the jvm bytecode I was generating for each statement and expression was valid bytecode and produced the proper output.

Overall, I think that test driven development is a fantastic way to develop a compiler and software in general. The tests set clear goals about what needed to happen and showed me when unexpected vs expected behavior was happening. For instance, if a string token was being produced rather than an integer token, test driven development would both show me that error and allow me to see how it was caused. In conclusion, I found this method very useful and effective during development.