1 CSCI 468 Capstone Portfolio - Charis Liddle

1.1 Section 1 - Program

My program source code (source.zip) can be found in the same directory as this report.

1.2 Section 2 - Teamwork

Our teams workload split was around 60-40. My primary role on the team was to write the code. This included the tokenizer, parser, and bytecode generator for the Catscript compiler. While I did that, my team member wrote the documentation for the project. This included descriptions of every syntactical element in Catscript, as well as a tutorial for each feature in Catscript. Working together, we both gained a good understanding of how compilers and programming languages work. My team member also wrote three tests for me to ensure my code worked properly.

1.3 Section 3 - Design Pattern

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = cache.get(type);

    if(listType == null) {
        listType = new ListType(type);
        cache.put(type, listType);
    }

   return listType;
}
```

In the code shown above, implemented within the Catscript type system, I used a design pattern called memoization to aid with determining the types of list objects. Memoization is a pattern designed to conserve memory by storing unique values so that they only need to be computed once.

This function utilizes memoization by storing each unique list type in a hashmap, so that whenever it encounters that list type later in the code, it can find that type in constant time and using up no additional memory. This is especially useful for list types, since lists can contain any type of component, including other lists.

1.4 Section 4 - Technical Writing

1.4.1 Catscript Programming Language

Introduction

Catscript is a relatively simple scripting language. Here is a basic example:

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

Features

Catstript has an array of features, including multiple different statements and expressions.

For Loops

For loops are a useful tool for iterating through lists of expressions. You will specify an identifier (item) and utilize a pre-defined list (myList). Within the loop, you are welcome to include any additional statements you would like to run each iteration.

```
for ( item in myList ) {
    print(item)
}
```

If Statement

The if statement is used to selectively run code. Within the parentheses you can insert an expression that evaluates to a boolean value. If the expression evaluates to true, the code within will run. Additionally, you can chain if statements together with optional else ifs and/or a final else. Only one section of an else if chain will run, and the else will run if no other statements evaluate to true.

```
if ( a == true ) {
    print("foo")
}
```

Print Statement

A print statement will print whatever is within its parentheses. If it is a primary expression, the printout will be the value of the expression. Otherwise, the output will be the result of the expression.

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

Variable Statement

The variable statement allows you to save values within a string based identifier. This identifier can then be used within any future code, as long as it is in the same scope.

```
var a = 1
var c = false
```

They can also be hardcoded for any type.

```
var b: string = "foo"
```

Assignment Statement

The assignment statement is used for changing or reassigning a previously declared variable

```
var a = 2
a = a + 4 // a = 6
a = 1 // a = 1
```

Function Call Statement

The function call statement in used for invoking a function. In this example, the function name is fooBar and it takes an int, a boolean value, and a string respectively.

```
fooBar(1, true, "HelloWorld")
```

Equality Expression

Equality expressions are used for verifying that two values are equivalent. This is most commonly used in if statements, but has application anywhere a boolean value is needed.

```
if (a == b) {
    print("true")
}
else if (a != b) {
    print("false")
}
```

Comparison Expression

Comparison expressions are similar to equality expressions, but have the ability to check the size relationship of int values.

```
if (a > b) {
    print("greater")
}
else if (a < b) {
    print("less")
}
else if (a >= b) {
    print("greater or equal")
}
else if (a <= b) {
    print("less or equal")
}
```

Additive Expression

The Additive expression can do addition or subtraction on int values. The output is the result.

```
var a = 1 + 1 // 2
var b = 2 - 1 // 1
```

You can also concatenate strings with the '+' operator.

```
var a = "Hello" + " World!" // "Hello World!"
```

Factor Expression

The factor expression multiplies or divides int values. The output is the result.

```
var a = 1 * 2 // 2
var b = 4 / 2 // 2
```

Unary Expression

The unary is used for negating values. There are two main applications of this, negating boolean values and negating int values.

```
if (not a == b) { // if a == b results in false, inner code will run
    print("a does not equal b")
}

var c = 1
print(-c) // -1
```

Primary Expression

This is a list of all primary expressions included in Catscript.

```
print(a) // identifier
print("Hello World") // string
print(1) // int
print(true) // boolean
print(null) // null
print(foo()) // function call
```

1.5 Section 5 - UML

In the UML sequence diagram displayed below, I have illustrated the parsing process for a variable statement in Catscript (var x : int = 6 / 3). The parser begins in the method parseProgram, and proceeds to parse the statement by calling successively more specific methods.

When it gets to parseVariableStatement, it first parses the type expression (: int) to determine the explicit type of the variable. Then it calls parseExpresion,

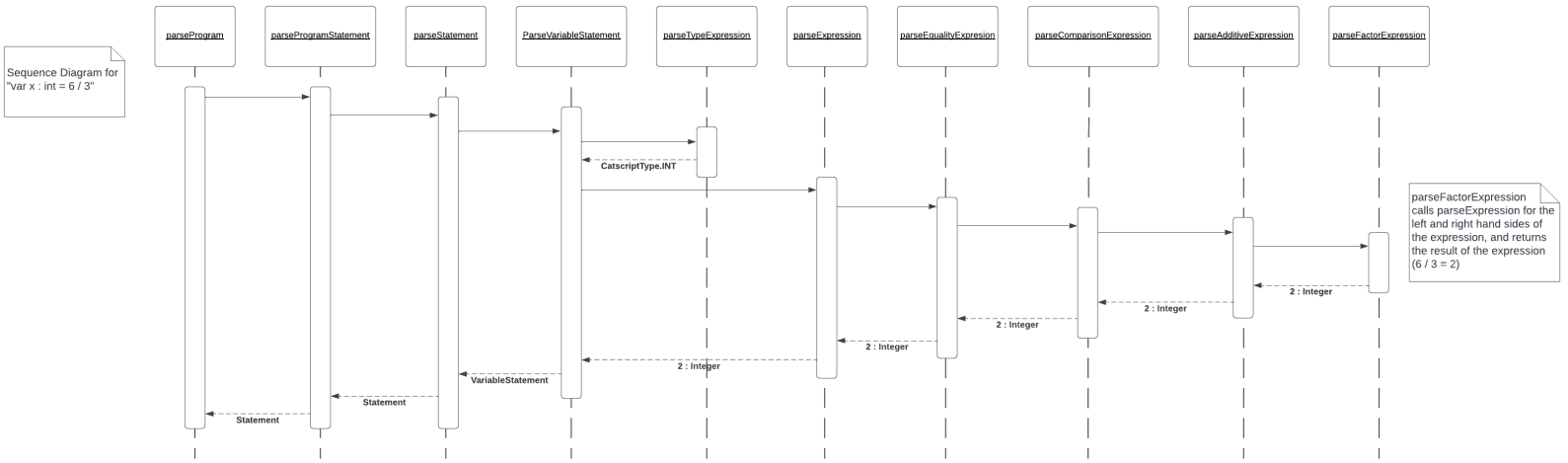

Figure 1: UML Sequence Diagram of parsing a variable statement in Catscript

which follows the recursive descent pattern all the way down to parseFactorExpression, where it parses the expressions on the left and right hand side (6 and 3), then returns the result of the division (6 / 3 = 2).

That value is returned through the stack of methods back up to the statement parser, and assigned as the variables value. Finally, the resulting parsed variable statement is returned back to the parent program.

1.6 Section 6 - Design Trade-offs

One major design trade-off in this project was the decision to write a recursive descent parser instead of using one of the many parser generator tools that are available. This decision was a good choice for a few reasons. First, it allowed me to dig into how the parsing actually works, and really understand what is happening in the code. Second, the code I wrote is much more sophisticated and less complicated than the code that a parser generator would write, making it easier to debug and find errors in. Lastly, writing my own recursive descent parser makes it easier for me to use my knowledge of compilers in practical applications that typically do not require all of the complexities imposed by parser generators.

1.7 Section 7 - Software Development Life Cycle Model

For this project, we used Test Driven Development (TDD). This means that the requirements of the system were compiled into nearly 200 separate test cases that ensured that each of those cases were processed and compiled correctly by the system. This helped me to see that my code was working correctly, both when I was finished, and as I was working through each section of the project.

These tests were designed to guide me through each section of the project.

They built on each other as I went, which ensured that I didnt skip ahead too far in the project. This helped me see how different sections of the project relied on each other, and gave me a good understanding of the process that I was implementing.

In addition to helping me see that my code was worked correctly, these tests were a good progress tracker as I worked through the project. I was able to use the tests to know what needed to be implemented next. It was extremely satisfying to see more and more tests pass as I made progress!