

Emily Foss

CSCI 468 - Compilers

Capstone Documents

28 April 2023

## Section 1: Program

See the GitHub directory labeled “capstone” for a zip file of the source code for the project.

## Section 2: Teamwork

Team member 1 was the primary development engineer on the project and was in charge of implementing the Catscript compiler features. Team member 2 was the testing and documentation engineer who was responsible for providing three additional tests as well as the technical documentation guide that details Catscript features in section 4.

## Section 3: Design Pattern

A design pattern that was implemented within this project was the memoization pattern. This pattern helped to optimize without having to alter the source code a great deal. Memoization is effectively caching and I implemented it to save the computation required to new up a list type when it is called with the same component time many times. This is done in the CatscriptType.java file within a method called getListType(). A snippet of this code can be found below:

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);
    if (listType == null){
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return listType;
}
```

## **Section 4: Capstone Guide**

### **Introduction**

Catscript is a simple scripting language developed by Montana State University, and is utilized in the senior capstone course. Some of the programming language's features include static typing, recursion, and lexical scoping. There are numerous other features such as plus operator overloading, all of which will be gone over in detail throughout the rest of this section.

### **Type Literals**

The six different data types the Catscript supports are integers, booleans, strings, lists, objects, and null (the keywords being int, bool, string, list, object, and null respectively). These types work very similarly to the same data types in Java. Catscript is statically typed, so once a variable is declared with a given type and name, that name cannot be used for a variable with a different type at any point in the script. You do not need to directly declare a variable type, as Catscript will infer what that variable's type is. So both of these examples are valid ways to declare a variable of type integer in Catscript

```
var x = 10  
var x : int = 10
```

### **Variable Statement**

A variable statement (as shown in the example above) declares and defines a variable for later use in the program. These consist of the 'var' keyword followed by a variable name, an equal sign, and a value. As mentioned above Catscript will infer what the variable's type is, but a specific type can be set by providing a colon and a type after the variable's name. If the type is incompatible with the value set, an error will be produced. For example:

```
var x : int = "10"
```

This statement will produce an error, as the quotation marks make the 10 a string, which is incompatible with integers.

### **Assignment Statements**

Once a variable is declared, its value can then be changed as long as the typing remains consistent. This is done with an assignment statement. If the typing is not consistent, then it will error out similarly to the variable statement. These statements consist of the variable name, followed by an equal sign, followed by the new value. For example:

```
x = 15
```

This sets the value of x from whatever it was to 15. If x was not properly declared with a variable statement, then it will error out. This statement still works even if the original value of x was 15.

## **Expressions**

An expression is a string of code that can be evaluated into a single value. These come in a large variety of types which we will now get into.

### **Equality Expression**

Equality expressions determine if two values equal one another. They consist of one value, followed by an equality operator, followed by another value, and they evaluate into boolean literals. The different equality operators used in catscript are equals and bang equal (or not equal), and their symbols are == and != respectively. For example:

3 == 4 evaluates to false  
3 != 4 evaluates to true  
"x" == "x" evaluates to true

### **Comparison Expression**

Comparison expressions similarly also evaluate into boolean literals and have the same structure (using comparison operators instead of equality operators), but these are determined based on comparative values. For the most part they are used to determine whether one value is greater than or less than the other. There are four comparison operators: greater than, greater than or equal to, less than, and less than or equal to (>, >=, <, <= respectively). For example:

3 > 4 evaluates to false  
3 < 4 evaluates to true  
3 <= 4 evaluates to false

### **Additive Expression**

Additive expressions handle addition and subtraction operations, and evaluate to an integer value. The structure is an integer, followed by a + or a -, followed by another integer. For example:

3 + 4 evaluates to 7  
3 - 4 evaluates to -1

It is also important to note that the + operator has also been overloaded for string concatenation. For example:

"banana" + "boat" evaluates to "banana boat"

### **Factor Expression**

Factor expression works almost identically to additive expression, it just works with multiplication and division instead (\* and / respectively). Neither of the factor operators have been overloaded to work with strings. For example:

4 \* 2 evaluates to 8

4 / 2 evaluates to 2

"banana" \* "boat" throws an error

### **Unary Expression**

Unary expressions, unlike all of the other expressions, involve an operator and only one other value. These are used to turn an integer from negative to positive (or vice versa) or a true to a false (or vice versa). The two unary operators are '-' (used for ints) and 'not' (used for bools). For example:

- 3 evaluates to a negative 3

not true evaluates to false

### **List Literal Expression**

A list literal expression is an object that contains a series of other objects. These are composed of a series of objects of values separated by commas, and enclosed in square brackets. It is possible to have a list of lists. For example:

[2, 3, 5, 7] is a list of integers

### **Expressions Continued**

It is important to note that multiple different expression types can be chained together, and utilizing parenthesis can ensure what operators you want to evaluate first. For example:

(3 + 7) \* 2 evaluates to 20

(3 + 4) == (4 + 3) evaluates to true

not not false evaluates to false

### **Print Statement**

The print statement takes an expression of some kind, evaluates it, and displays it in the console. These statements are composed of the keyword print, followed by an expression in parentheses. This can be a variable, a given value, or any other expression. For example:

```
var x = 10 print(x)
print("10")
print(3+4+3)
```

All of the above print statements will display a '10' in the console.

### **For Statement**

For statements have a body of code that is run a number of times given by an expression. This expression also often declares a variable whose value changes based on which iteration of the loop the statement is currently in. The syntax of a for loop is the keyword for, followed by an iterable expression in parentheses, followed by a body of code in curly brackets. For example:

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

In the for statement above, the print statement will execute three times, and will print off each value in the list in order. So it will print 1, then 2, then 3, and then the for statement will be complete.

### **If Statement**

If statements also have their own body of code that is only executed if conditional requirements are met. They consist of the if keyword, followed by an expression that outputs a boolean value, which is then followed by a body of code. If the expression evaluates to True, then the body of code is executed. If not, then it is skipped over. If statements can also have an else statement immediately after. This also has a body of code, but this body is only executed if the expression in the if statement evaluates to false. For example:

```
if( 3 == 5 ){ print(5) }
else { print(10) }
```

In the above case, the '3 == 5' would evaluate to False, so the body of the if statement would not execute, and the body of the else statement would execute instead, printing out '10' to the console.

### **Functions**

Functions are blocks of code assigned to a variable name, so that they can be executed repeatedly throughout different parts of the program as their own statements. There can be parameters associated with the function that work as variables whose scope is limited to that function, the values of which are set when the function is called. The form of these are the word 'function' followed by a variable name (used to call the function elsewhere), and open and close parenthesis where parameters will go if included (which take the form of a variable name, a

colon, and a variable type. Multiple parameters are separated by commas. Then a function body is declared surrounded by curly braces. For example:

```
function addFive( x : int ) {  
    print(x + 5)  
}  
  
addFive(5)
```

The first block is the declaration of the function, and the next line after is calling that function. In this case the parameter 5 is passed to the function which becomes the value for x, so the print statement ends up displaying 10 to the console.

### **Return Statements**

Return statements are used if you want your function to create a value when it is called. This is done by adding the keyword 'return' to the end of a function body, followed by whatever value you want returned. When a function is declared, the type the function returns will also need to be declared which is done by adding a colon followed by the type after the parenthesis and before the curly brackets. For example:

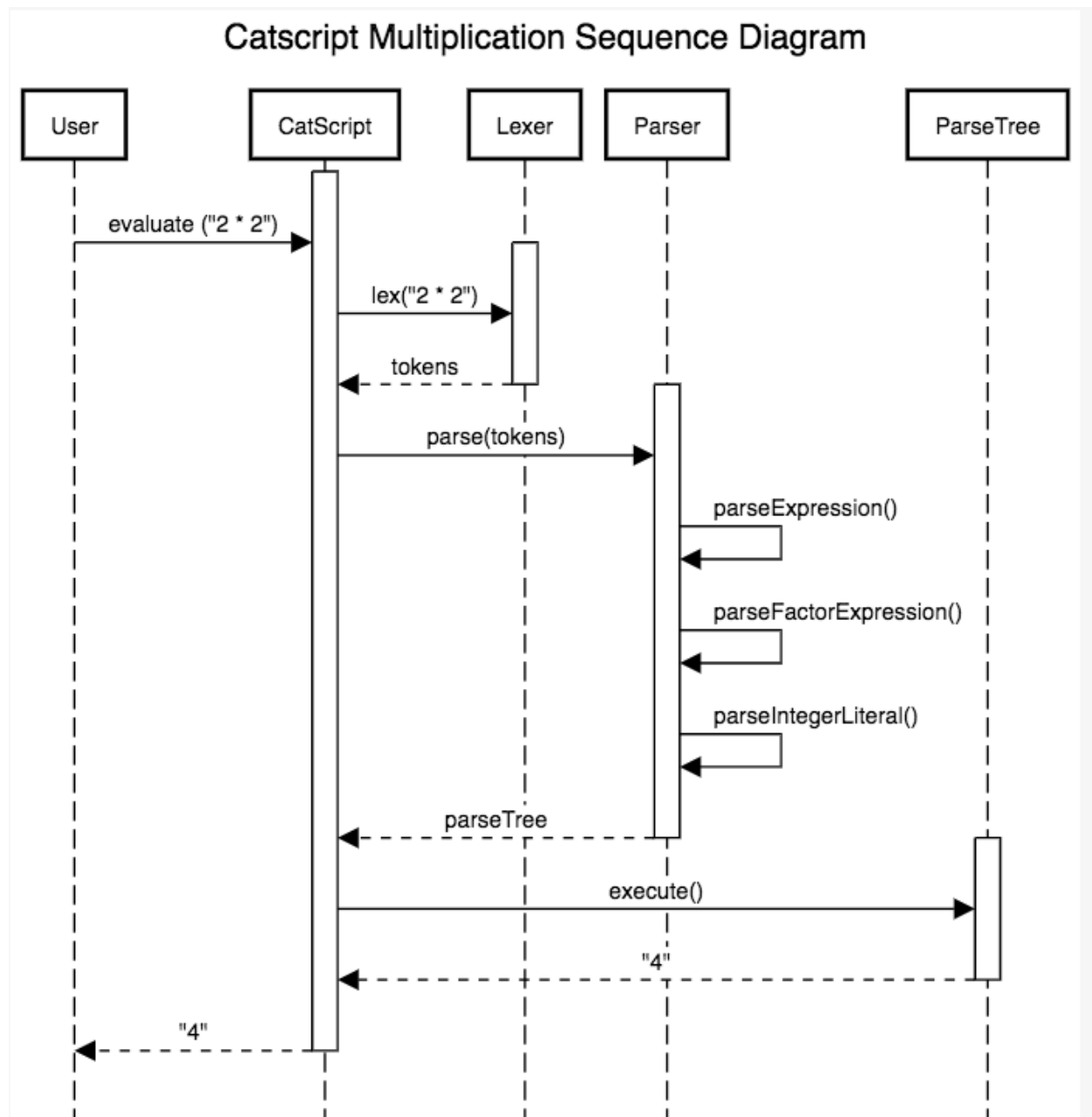
```
function addFive(x : int) : int { return x + 5 }  
  
var num = addFive(5)
```

Similarly as the example for functions, this function takes in an integer and adds 5 to it. Instead of printing the result to the console however, it returns the value so that it can be assigned to a variable outside of the function. This also works outside of variable statements. For instance:

```
print(addFive(10))
```

This combined statement would print '15' to the console.

## Section 5: UML Sequence Diagram for Multiplication Parse Element



## Section 6: Design Trade Offs

The most significant design trade off to be considered with this project is the fact that this compiler was written completely by hand rather than built with the aid of a parser generator. For the purposes of learning the inner workings of a recursive descent parsing algorithm and how it can be implemented in a compiler, we were much better served by the approach chosen. Parser generators allow you to simply define a grammar and plug it into the tool you choose and it spits

out source code. This offers a faster turn around and is simpler, but would keep a lot of what is going on under the hood concealed. We had an entire 15 weeks with a lot of the framework code provided, so churning out code quickly was not a primary concern. The end result of a parser generator is often more easily maintainable, but that does not really apply in this case, since I do not expect many of the students in this course will be revisiting this code very often. A good parser generator will also usually build code that is faster, but again, this advantage does not matter much to us, since Catscript is a hypothetical programming language in the first place.

## **Section 7: Software Development Lifecycle Model**

The software development process used for this project was Test Driven Development. A suite of tests was provided and code was written to fulfill the requirements of each test. This model is very ubiquitous in industry and is utilized at the company I intern for. TDD results in robust code. The team knew exactly what needed to be completed for the code to meet the set standards and little to no effort was wasted. Debugging was easy with the tests. The downfalls of this method were some of the tests passing despite the implementation being slightly incorrect. This caused some problems later on down the line that were difficult to identify since it required backtracking down many possible avenues to discover the problem. To remedy this, one would need to write more tests. Test suites can blow up fast and it is tricky to perfect tests such that nothing is missed. Passing a test can give a developer a false sense of security that no issues will arise with the code.