

Section 1:

Program

<https://github.com/COLEREIMER/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

Section 2:

Teamwork

For the teamwork portion of this project, I was the primary engineer and my partner, Gage Hillyard, was the documentation and testing engineer. Gage wrote three test cases for my compiler code along with a comprehensive guide to the Catscript language. Gage's tests were all tailored towards the execution section of my compiler. They were designed with the intent of testing the integrity and functionality of my code. The Catscript documentation is all under the section four technical writing portion of the portfolio. All of Gage's supplied test cases can be seen in the code chunks below.

```
@Test
void varInsideFunctionWorksProperly() {
    assertEquals("60\n", executeProgram("function capstone() : int {\n" +
        "    var x = 30\n" +
        "    var copy = 30 * 2\n" +
        "    return copy\n" +
        "}\n" +
        "print( capstone() )\n"));
}

@Test
void localVarStatementsWorkProperly() {
    assertEquals("I like the number 1\nI like the number 2\n",
executeProgram("for( x in [1, 2] ) {\n" +
        "    var y = x\n" +
        "    if(y == x){\n" +
        "        print(\"I like the number \" + y)\n" +
        "    }\n" +
        "}\n"));
}
```

```

    }

    @Test
    void forStatementWorksProperly() {
        assertEquals("2\n3\n4\n3\n4\n5\n4\n5\n6\n5\n6\n7\n",
executeProgram("for(x in [1, 2, 3, 4]) {\n" +
                "for(y in [1, 2, 3]) {\n" +
                "    print(x + y)\n" +
                "}\n" +
                "}")");
    }

```

Section 3: *Design Pattern*

The design pattern that I chose to implement in my compiler code was “memoization”. The purpose of adding this design pattern was to save myself the computation of having to constantly re-initialize the list type variable. This design pattern was implemented in the getListType method of the CatscriptType class in the parser section of the compiler code. The code for the design pattern is all included and boxed in the code chunk below. This design pattern efficiently verifies whether or not a list type already exists. Only when it does not exist, the pattern creates a new list type and adds it to the hashmap. This approach eliminates the need to repeatedly reinitialize list types that are already stored in the hashmap.

```

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:

Technical Writing

Catscript Guide

This document is a comprehensive guide for the Catscript language.
Written by Gage Hilyard for Cole Reimer.

Introduction

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

```
var x = "foo"  
var y = "bar"  
print(x + y)
```

Features

Syntax

Catscript syntax is very simple. Unlike Java, semicolons are not needed to end a line. Source code can be entirely written on a single line if all statements are grammatically correct.

Inline comments are also familiar. Here is an example:

```
// This is a comment. Anything written past the double forward slashes  
// will be ignored when compiled.  
var x = 1  
print(x + 1)
```

Variables and Lists

Initializing variables in Catscript is very simple. Variables of type integers, strings and bools as well as lists can be initiated with or without an explicit type. Variables of type “Object” can be assigned to any type.

Variable statements are constructed as followed:

```
var variable_name ( : type ) = value
```

```
var variable : int = 406
```

Lists are initialized very similarly to other variables:

```
var variable_name ( : list<type> ) = [comma, separated, values]
```

```
var list : list<int> = [4, 0, 6]
```

Here are some examples of how variables are initialized with or without an explicit type:

```
var x : int = 1
```

```
var y = 1
```

```
var str : string = “hello”
```

```
var str2 = “world”
```

```
var bool : boolean = true
```

```
var bool2 = false
```

```
var obj : object = 1
```

```
var obj2 = “value”
```

```
var obj_list : list<object> = [“montana”, 406, true]
```

```
var int_list = [1, 2, 3]
```

```
var null_variable = null
```

Variables can be reassigned to another value as long as they are the same type unless the variable is initialized as an object.

Here are some examples:

```
var x = 1  
  
var y = 2  
  
x = 3  
  
x = y  
  
var obj : object = "asdf"  
  
obj = 10  
  
obj = true
```

Math

Catscript can do simple arithmetic, it can add, subtract, multiply, and divide integers. Catscript also supports negative integers and parentheses as well.

Expressions are constructed as followed:

```
addition 1 + 1  
  
subtraction 1 - 1  
  
multiplication 1 * 1  
  
division 1 / 1  
  
var math = 1 - (-1) + ((8/2)*4)
```

Print Statements

Print statements are easy to use and can concatenate strings and solve expressions. One thing to note is that the lines always end with a new line.

Here are some examples:

```
var x = "hello"  
var y = " world"  
var z = 406  
print("This is a Catscript Program: " + x + y + z)
```

The above print statement would output "This is a Catscript Program: hello world406".

```
var x = 1  
var y = 2  
print("1 + 2 = " + (x + y))
```

The above print statement would output "1 + 2 = 3".

If/Else Statements and Equality Expressions

If/Else statements are similar to Java if/else statements and just require a true or false case.

Here is an example:

```
var case = true  
  
if(case) {  
    print("This is true")  
}
```

```
} else {  
  
    print("This is false")  
  
}
```

If/Else statements can be combined with equality expressions as well. Here are some examples of equality expressions:

```
a > b //greater than  
a >= b //greater than or equal to  
a == b //equal to  
a != b //not equal to  
a <= b //less than or equal to  
a < b //less than
```

A realistic if/else statement may look like this:

```
if (variable_a >= variable_b) {  
  
    print("true")  
  
} else {  
  
    print("false")  
  
}
```

Booleans can also be negated with the 'not' keyword:

```
var bool = false  
  
if(not bool) {  
  
    print("This will print")  
  
} else {  
  
    print("This will not print")  
  
}
```

```
}
```

For loops

For loop statements require a list to iterate over. A common case would be to iterate over an integer list from 1 to an upper range. Here is an example:

```
//initializing a list in the for loop  
  
for (x in [1, 2, 3]) {  
  
    print(x)  
  
}
```

```
//initializing a list outside of the for loop  
  
iterList = [4, 5, 6]  
  
for (x in iterList) {  
  
    print(x)  
  
}
```

This would print the numbers 1 through 6 on their own lines.

Functions

Defining functions in Catscript is simple. Functions return types can be any of the variable types or void. Functions arguments may also have an explicit type or none at all.

A function declaration is constructed as follows:

```
//function with return type and argument types  
  
function function_name(arg1 : type, arg2 : type) : return_type {
```



```

var variable_of_return_type = value

return variable_of_return_type
}

//function with no return type or argument types
function function_name(arg1, arg2, arg3) {

    print(arg1 + arg2 + arg3)
}

```

Here is an example of a function that doubles and prints numbers from a given integer list:

```

function double (list : list<int>) {

    for(x in list) {

        print(x*2)

    }

}

```

To call a function, you use the function's name and any arguments it may have. For the previous function, it would be called as follows:

```

int_list = [2, 0, 3]

double(int_list)

```

Functions in Catscript can also be called recursively.

```

function recursive(x) {

    print(x)

    if(x > 0) {

        recursive(x - 1)

    }

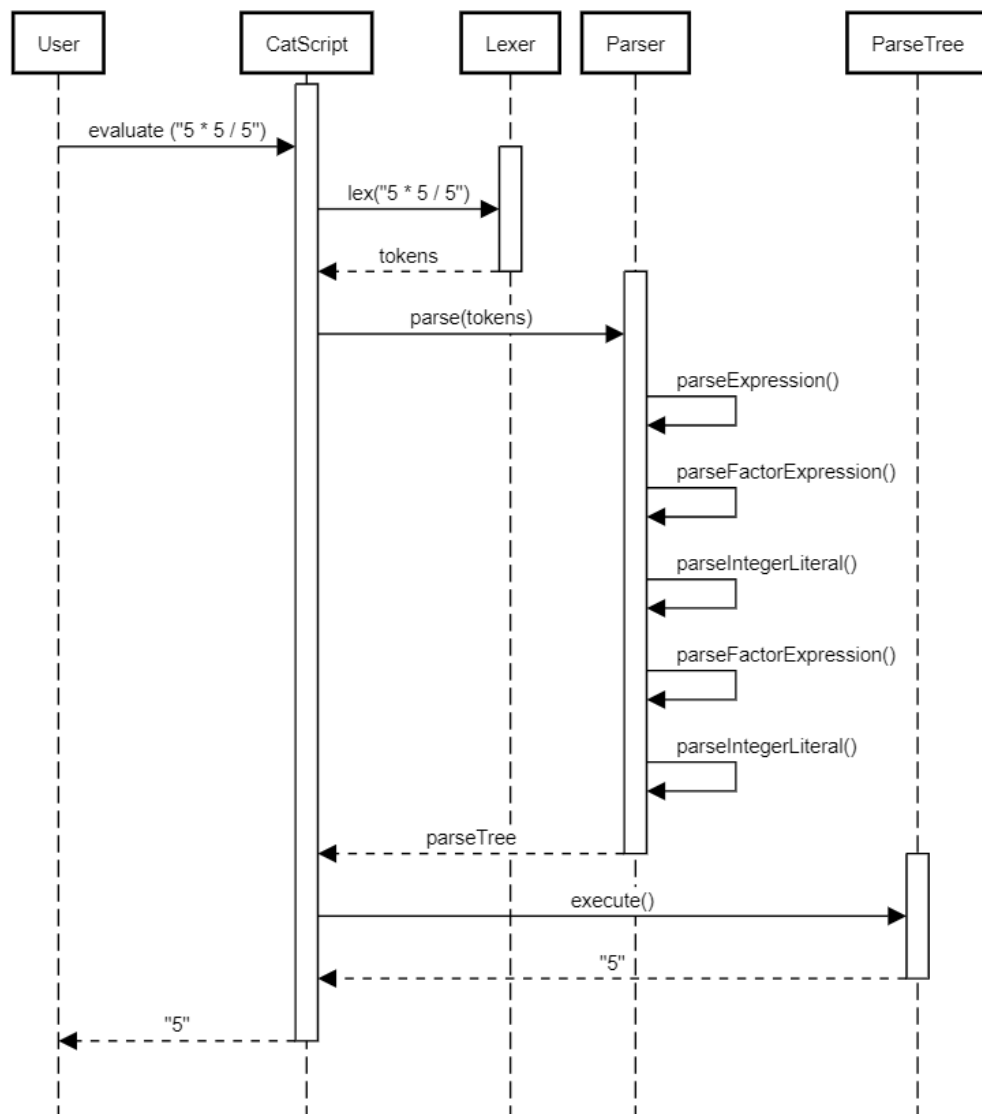
}

```

```
}  
}  
recursive(10)
```

Section 5: *UML*

Catscript Factor Sequence Diagram



This sequence diagram is used to demonstrate the evaluation of a simple factor expression. The user sends an expression that starts in evaluation and then Catscript uses “lex”, a way of scanning the expression, to break the evaluation into subsequent tokens and returns it to the Catscript program. From there, we use the parser to parse through them starting with the parse expression function. The parse expression function uses recursive descent to ultimately parse the left and right sides of both of the equations and then returns it to the Catscript program to execute. The parse tree class uses the execute function to return the value to the Catscript class which outputs the value back to the user.

Section 6:

Design Trade-Offs

The main design tradeoff that I’ve decided to focus on comes from the fact that we chose to write our parser by hand instead of simply using a parser generator. The various benefits of writing the parser by hand all stem from the fact that we are able to control the ultimate look and implementation of the code. Specifically, in our Catscript parser we chose to implement a recursive-descent styled algorithm. A recursive-descent algorithm is a simple yet elegant way to write a parser in such a way that makes it obvious how the grammar works. This approach is a much more intuitive way to implement code because the retention of the materials is the most important aspect of the project. Although there exist many benefits to writing a parser by hand, hand-written parsers can often be very time consuming and error-prone.

Parser generators are a tool that can generate the code for parsers based on the specific grammar of the language. Parser generators help to increase both the speed and accuracy of the generator while also making it easy to maintain the code as programming languages oftentimes like to evolve. The drawbacks of parser generators all come from the fact that they have limitations when it comes to the customization of the code. For

example, they could have trouble trying to integrate the parser code into a previously existing code base. When trying to decide between the implementation of generated or handwritten parsers, it's best to take into account the ultimate goal of the project. In this class, it was apparent that writing the parser by hand would be most beneficial.

Section 7:

Software Development Life Cycle Model

The test-driven development model proved to be highly beneficial to the development of my code. By looking at the tests before even starting to code each portion, I was able to get a solid understanding of what the code would be producing along with a general grasp of how and where to begin. Furthermore, the test-driven model helped to catch errors early on in the code so that they wouldn't be a problem as the project continued to build upon itself. This was especially helpful for this project because every checkpoint would rely on the proper implementation of the section preceding it. Lastly, when the code wasn't outputting the correct value or response, I was able to compare my output with the output that the test was expecting. This made it a lot easier to debug and fix any issues in the code. Overall, the test-driven model greatly benefited my personal development and retention of the code by emphasizing testing and enabling me to better understand/debug the code.