Section 1: Program

Please include a zip file of the final repository in this directory.

Section 2: Teamwork

Member 1 was responsible for the implementation of the tokenizer and parser portion of our Tree-Walk Interpreter as well as the evaluate and compile functions that allow Catscript to run on a Spark server or to compile into bytecode that can be ran on the JVM. In addition to this, Member 1 had to make sure each expression and statement validated properly for a bug-free execution of the code. Member 1 also supplied Member 2 with tests that targeted linking specific functions together, as shown below:

```
@Test
void testingListTypes(){
    VariableStatement var = parseStatement("var x : list<int> = [1]");

    assertEquals(CatscriptType.getListType(CatscriptType.INT),
var.getExplicitType());

    VariableStatement var1 = parseStatement("var x : list<bool> = [true]");
    assertEquals(CatscriptType.getListType(CatscriptType.BOOLEAN),
var1.getExplicitType());

    VariableStatement var2 = parseStatement("var x : list<string> = [\"foo\"]");
    assertEquals(CatscriptType.getListType(CatscriptType.STRING),
var2.getExplicitType());

    VariableStatement var3 = parseStatement("var x : list<object> = [1]");
    assertEquals(CatscriptType.getListType(CatscriptType.OBJECT),
var3.getExplicitType());
}

@Test
void testingVars() {
    assertEquals("21\n", executeProgram("var x : int = 3 * (1 + 9 / 3) - (- 9)\n"
+
       "print(x)"));
    assertEquals("21A\n", executeProgram("var x : string = 3 * (1 + 9 / 3) - (- 9)
+ \"A\"\n" +
        "print(x)"));
    assertEquals("A21\n", executeProgram("var x : string = \"A\" + (3 * (1 + 9 /
3) - (- 9))\n" +
        "print(x)"));
}

@Test
void testingElseIfStatements() {
    assertEquals("21\n", executeProgram("function foo(x : int) : int {\n" +
```

```
            "if (x < 21) {\n" +
                "return x * 2}\n" +
            "else if (x == 21) {\n" +
                "return x}\n" +
            "else {return -x}" +
                "return 0}\n" +
            "var a : int = 21\n" +
            "print(foo(a))"
        ));
        assertEquals("40\n", executeProgram("function foo(x : int) : int {\n" +
            "if (x < 21) {\n" +
                "return x * 2}\n" +
            "else if (x == 21) {\n" +
                "return x}\n" +
            "else {return -x}" +
                "return 0}\n" +
            "var a : int = 20\n" +
            "print(foo(a))"
        ));
        assertEquals("-23\n", executeProgram("function foo(x : int) : int {\n" +
            "if (x < 21) {\n" +
            "return x * 2}\n" +
                "else if (x == 21) {\n" +
            "return x}\n" +
                "else {return -x}" +
            "return 0}\n" +
            "var a : int = 23\n" +
            "print(foo(a))"
        ));
    }
```

Member 2 was responsible for their own implementation of Catscript through a tokenizer, parser, tree-walk interpreter and a compiler. Member 2 also supplied Member 1 with multiple tests to ensure that Member 1's implementation was correct. This involved linking if/else statements together, often within functions to test additive and factor expressions in addition to string concatenation and whether functions evaluated properly. These tests are shown below:

```
    @Test
        void testingAdditiveAndFactorExpressionsInIfStatements() {
            assertEquals("32\n", executeProgram("function foo(x : int) {\n" +
                    "   if (x < 9 * 2 + 4 * 3) {\n" +
                    "       return x / 2\n" +
                    "   } else {\n" +
                    "       return x\n" +
                    "   }\n" +
                    "}\n" +
                    "var y : int = 32\n" +
                    "print(foo(y))"));
            assertEquals("true\n", executeProgram("function foo(x : string) {\n" +
                    "   if (\"foo bar\" == x + \" bar\") {\n" +
                    "       return true\n" +
```

```
                "   } else {\n" +
                "       return false\n" +
                "   }\n" +
                "}\n" +
                "var y : string = \"foo\"\n" +
                "print(foo(y))"));
        assertEquals("foo input: 10\n", executeProgram("function foo(x : string, y
: int) {\n" +
                "   if ( - y == (y - 30) / 2 ) {\n" +
                "       return x + \" input: \" + y\n" +
                "   } else {\n" +
                "       return false\n" +
                "   }\n" +
                "}\n" +
                "var a : string = \"foo\"\n" +
                "var b : int = 10\n" +
                "print(foo(a, b))"));
    }

    @Test
    void testingIfStatementsInIfStatements() {
        assertEquals("equal to 20\n", executeProgram("var x = 20\n" +
                "if (x < 20) {\n" +
                "   if (x < 10) {\n" +
                "       print(\"less than 10\")\n" +
                "   } else if (x >= 10) {\n" +
                "       if (x == 10) {\n" +
                "           print(\"equal to 10\")\n" +
                "       } else {\n" +
                "           print(\"greater than 10\")\n" +
                "       }\n" +
                "   }\n" +
                "} else if (x == 20) {\n" +
                "   print(\"equal to 20\")\n" +
                "} else {\n" +
                "   print(\"greater than 20\")\n" +
                "}\n"));
        assertEquals("greater than 10\n", executeProgram("var x = 15\n" +
                "if (x < 20) {\n" +
                "   if (x < 10) {\n" +
                "       print(\"less than 10\")\n" +
                "   } else if (x >= 10) {\n" +
                "       if (x == 10) {\n" +
                "           print(\"equal to 10\")\n" +
                "       } else {\n" +
                "           print(\"greater than 10\")\n" +
                "       }\n" +
                "   }\n" +
                "} else if (x == 20) {\n" +
                "   print(\"equal to 20\")\n" +
                "} else {\n" +
                "   print(\"greater than 20\")\n" +
                "}\n"));
        assertEquals("equal to 10\n", executeProgram("var x = 10\n" +
```

```
                "if (x < 20) {\n" +
                "   if (x < 10) {\n" +
                "       print(\"less than 10\")\n" +
                "   } else if (x >= 10) {\n" +
                "       if (x == 10) {\n" +
                "           print(\"equal to 10\")\n" +
                "       } else {\n" +
                "           print(\"greater than 10\")\n" +
                "       }\n" +
                "   }\n" +
                "} else if (x == 20) {\n" +
                "   print(\"equal to 20\")\n" +
                "} else {\n" +
                "   print(\"greater than 20\")\n" +
                "}\n"));
        assertEquals("less than 10\n", executeProgram("var x = 5\n" +
                "if (x < 20) {\n" +
                "   if (x < 10) {\n" +
                "       print(\"less than 10\")\n" +
                "   } else if (x >= 10) {\n" +
                "       if (x == 10) {\n" +
                "           print(\"equal to 10\")\n" +
                "       } else {\n" +
                "           print(\"greater than 10\")\n" +
                "       }\n" +
                "   }\n" +
                "} else if (x == 20) {\n" +
                "   print(\"equal to 20\")\n" +
                "} else {\n" +
                "   print(\"greater than 20\")\n" +
                "}\n"));
        assertEquals("greater than 20\n", executeProgram("var x = 40\n" +
                "if (x < 20) {\n" +
                "   if (x < 10) {\n" +
                "       print(\"less than 10\")\n" +
                "   } else if (x >= 10) {\n" +
                "       if (x == 10) {\n" +
                "           print(\"equal to 10\")\n" +
                "       } else {\n" +
                "           print(\"greater than 10\")\n" +
                "       }\n" +
                "   }\n" +
                "} else if (x == 20) {\n" +
                "   print(\"equal to 20\")\n" +
                "} else {\n" +
                "   print(\"greater than 20\")\n" +
                "}\n"));
    }

    @Test
    void testFuncCallsInFunctions() {
        assertEquals("100\n", executeProgram("" +
                "function doubleComparison(e : bool, f : bool) : bool {\n" +
                "   if (e == true) {\n" +
```

```
                "       if (f == true) {\n" +
                "           return true\n" +
                "       } else {\n" +
                "           return false\n" +
                "       }\n" +
                "   } else {\n" +
                "       return false\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "function foo(x : int, y : int) : int {\n" +
                "   var c : bool = x > 10\n" +
                "   var d : bool = y == 5\n" +
                "   if (doubleComparison(c, d)) {\n" +
                "       return x * y\n" +
                "   } else {\n" +
                "       return x + y\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "var a : int = 20\n" +
                "var b : int = 5\n" +
                "print(foo(a, b))"));
        assertEquals("15\n", executeProgram("" +
                "function doubleComparison(e : bool, f : bool) : bool {\n" +
                "   if (e == true) {\n" +
                "       if (f == true) {\n" +
                "           return true\n" +
                "       } else {\n" +
                "           return false\n" +
                "       }\n" +
                "   } else {\n" +
                "       return false\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "function foo(x : int, y : int) : int {\n" +
                "   var c : bool = x > 10\n" +
                "   var d : bool = y == 5\n" +
                "   if (doubleComparison(c, d)) {\n" +
                "       return x * y\n" +
                "   } else {\n" +
                "       return x + y\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "var a : int = 10\n" +
                "var b : int = 5\n" +
                "print(foo(a, b))"));
        assertEquals("23\n", executeProgram("" +
                "function doubleComparison(e : bool, f : bool) : bool {\n" +
                "   if (e == true) {\n" +
                "       if (f == true) {\n" +
                "           return true\n" +
```

```
                "       } else {\n" +
                "           return false\n" +
                "       }\n" +
                "   } else {\n" +
                "       return false\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "function foo(x : int, y : int) : int {\n" +
                "   var c : bool = x > 10\n" +
                "   var d : bool = y == 5\n" +
                "   if (doubleComparison(c, d)) {\n" +
                "       return x * y\n" +
                "   } else {\n" +
                "       return x + y\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "var a : int = 20\n" +
                "var b : int = 3\n" +
                "print(foo(a, b))"));
        assertEquals("24\n", executeProgram("" +
                "function doubleComparison(e : bool, f : bool) : bool {\n" +
                "   if (e == true) {\n" +
                "       if (f == true) {\n" +
                "           return true\n" +
                "       } else {\n" +
                "           return false\n" +
                "       }\n" +
                "   } else {\n" +
                "       return false\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "function foo(x : int, y : int) : int {\n" +
                "   var c : bool = x > 10\n" +
                "   var d : bool = y == 5\n" +
                "   if (doubleComparison(c, d)) {\n" +
                "       return x * y\n" +
                "   } else {\n" +
                "       return x + y\n" +
                "   }\n" +
                "}\n" +
                "\n" +
                "var a : int = 20\n" +
                "var b : int = 4\n" +
                "print(foo(a, b))"));
    }
```

Member 1 was responsible for 50% of the time spent on this project while Member 2 was responsible for 50% of the time spent on this project.

Section 3: Design pattern

Memoization is an optimization pattern that uses a cache of data types to prevent memory from filling with many versions of the same data type. With this cache, instead of having to create a new datatype each time a List is created we can instead have a single version of each type that we can reference when a new List is created, leading to at most only one of each type of list existing in memory at any given time. This is an easy way to optimize memory usage and was an easy implementation for our code, meaning that this pattern improved the efficiency of our code for very little more effort over hard-coding our list types. Below is the code we use to implement memoization.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    if (cache.containsKey(type)) {
        return cache.get(type);
    } else {
        cache.put(type, new ListType(type));
    }
    return cache.get(type);
}
```

In the above code we utilise a static hashmap to cache any new list types that need to be created, returning a reference to a single type instead of a new reference.

Section 4: Technical writing. Include the technical document that accompanied your capstone project.

Introduction

Catscript is a simple scripting langauge that supports primative data types (Integers and Booleans) as well as a few reference types (Strings, Objects, Lists). Catscript also allows for basic for loops, as well as basic logic control if/else statements. Catscript has no compilation rules for whitespace, and does not require the use of semicolons at the end of statements. Along with this, Catscript uses left hand associativity for mathematical expressions. Here is some small example code:

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

Features

Typing

Dynamic Typing

Catscript includes the feature of dynamic typing, this means that when a variable is created without an explicit type specified, the program will assign a type to it based on the first expression that is assigned to the variable. Once an expression has been assigned, the variable can no longer be assigned to another expression type.

```
var x = 10
x = (any other int value)
var y = "string"
y = (any other string value)
```

Explicit typing

Other than the dynamic typing feature, Catscript also offers the option to explicitly type variables. This will ensure that variables will not get assigned an undesired expression type.

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

Expressions

Expressions are objects in Catscript that produce values.

Type Expressions

A type expression in catscript is an expression that declares the data type of an object. These can be applied to things like variables and functions.

Data Types

String Literal Expression

String Literal expressions are objects that are a collection of text. Catscript uses the Java style string object.

```
"this is a string"
```

Int Literal Expression

Integer Literal Expressions in Catscript are 32 bit integers.

```
10
5
12
```

Boolean Literal Expression

Boolean Literal Expressions are true or false values.

```
true
false
```

List Literal Expressions

Catscript uses a Linked List style array, this can hold integers, strings, booleans, or null values.

```
[0, 1, 2]
[0, 1, 2]
```

Null Literal Expressiont

Catscript uses Java style null type.

```
var x = null
```

Addidtive Expression

Catscript has the feature of additive expressions. Additive expressions are expressions that contain the "+" and "-" operators. These expressions come in the form of a factor expression + or - another factor expression.

```
1 + 1 -> 2
(1 + 1) - (3 * 2) -> -4
```

Factor Expressions

Factor expressions are similar to additive expressions. They allow for multiplication through the "*" operator and division through the "/" operator.

```
3 * 2 -> 6
(5 * 4) / 2 -> 10
```

Parenthesized Expressions

Parenthesized epxressions are expressions that are embedded within parenthesis. This will allow the expression within to be evaluated as a whole before anything else is done to the values within the expression.

```
(expression goes here)
(2 + 1)
(5 - 3) / (2 * 1)
```

Comparison Expressions

Comparison expressions in Catscript are expressions that use the ">", "<", ">=" and "<=" operators to compare the given values on the left and right side of the operator, and produce a boolean value.

```
1 < 2 -> true
45 >= 4 -> true
32 > 49 -> false
33 <= 33 -> true
```

Equality Expressions

Similar to comparison expressions, equality expressions use the "==" and "!=" operators to produce boolean values.

```
1 == 2 -> false
1 != 2 -> true
```

Unary Expressions

Unary expressions are expressions used to negate the boolean value or integer value within an expression. These use the "not" operator for boolean values and the "-" operator for integer values.

```
not (5 > 3) -> false
- (10 - 5) -> -5
```

Proper Order of Operations

Catscript follows the mathematical rules of PMDAS, in order from left to right. The exception to this rule is exponents, this is because there is no supported exponent operand in Catscript. Unary operators such as "-" and "not" are executed from right to left to ensure that values are properly negated.

String Concatenation

With additive expressions, Catscript allows for the concatenation of a string to a string, integer or a null value using the '+' operand.

```
var w = 1
var x = "foo"
var y = "bar"
var z = null
w + x -> "1foo"
x + y -> "foobar"
y + z -> "barnull"
```

Function Call Expressions

Declared functions can be called from anywhere in the program using function call expressions.

```
functionName1(variable1, variable2, ...)
var x = functionName2(variable1, variable2, ...)
```

Identifier Expressions

Identifier expressions are expressions that use a keyword created by the user called an Identifier to access functions or variables created in the program.

```
var x = 23
x -> 23
```

Statements

For Statements

Catscript offers the functionality of for loops to repeat a set of instructions. For statements take a list and iterate over each item in the list which are accessed through a supplied identifier value.

```
for (IDENTIFIER in [a, b, c]) {
    (statements to be repeated)
}
```

Variable Statements

Using the "var" operator, Catscript allows the user to create variables. These variables can use a type expression to assign a data type to them. If no data type is specified, then the program will use type inference with the first value assigned to the variable to set the variable type.

```
var x : int
var y : string
```

```
var z : bool
var a : list<int>
var b
```

Assignment Statements

In catscript you can assign an expression to a variable name using assignment statements with the "=" operator.

```
var x : int = 10
x = 11
x = 12

var y : object
y = "foo"
y = "string"
```

Object

Catscript uses Java style Object type that can take integers, strings, booleans or lists.

```
var x : object = [1, 2, 3]
var y : object = "foo"
```

List Types

Declaring the list of explicit type object will allow you to store values of differing types within the list. If you do not explicitly type it as object, and the list has differing types in it, then dynamic typing will assign it to an object list.

```
var x = [1, true, "foo"]
var y : list<object> = [1, true, "foo"]
```

Function Declaration Statements

In Catscript, you can define a set of instructions to preform by declaring a function. Function declarations are used to either modify variables, or return a value.

Void Functions

Void functions are functions with the void declared data type, these functions do not need return statements.

```
function functionName(variable1 : TYPE, variable2 : TYPE, ...) : void {
    (instructions go here)
}
```

Typed Functions

Typed functions are used to return a value of a certain specified data type. These functions require that a return statement is in each possible branch of the function.

```
function functionName(variable1 : TYPE, variable2 : TYPE, ...) : TYPE {
    (instructions go here)
    return(VALUE or VARIABLE)
}
```

If Statements

If statements in Catscript are reliant expressions that produce boolean values such as "not", "<", "<=", ">", ">=", true, false, "==" and "!=". If statements must have an initial if branch, then can have optional else if branches or one optional else branches.

```
if (5 < 3) {
    (statements)
} else if (not false) {
    (statements)
} else if (x >= y) {
    (statements)
} else if (a != - b) {
    (statements)
} else {
    (statements)
}
```

Print Statements

Print statements allow the program to print any object to the console.

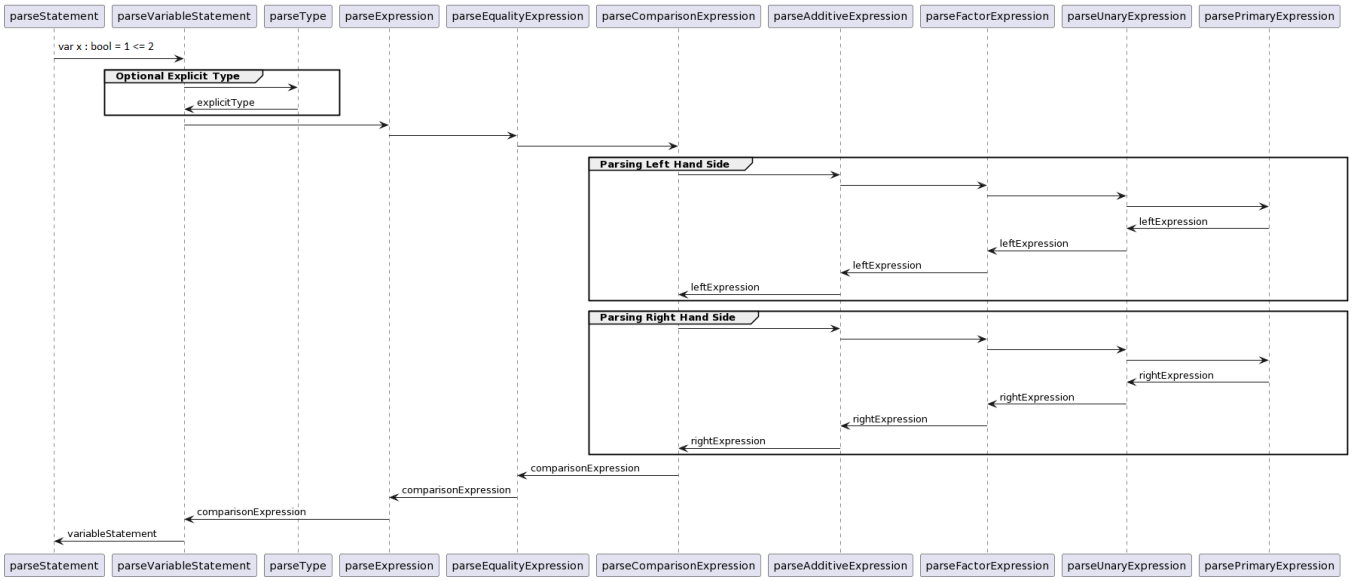
```
print(1)
print("foo")
print(variableName)
```

Return Statements

Return statements allow the declared functions to return some value within the expression, or allow for the program to break out of the current function.

```
return(1)
return("foo")
return(variableName)
```

Section 5: UML.



This image shows the parse sequence of the statement var x : Bool = 1 <= 2 . When the statement is first parsed when parseStatement is called, the first thing that happens is we try to match the token with a statement. In this case, the "var" token is matched and parseVarAssignment is called. Since we explicitly type the variable we then call parseType which will return the explicit type of the variable.

From here we have to parse the expression of the variable statement using parseExpression. This function flows down the hierarchy of our recursive descent until we reach the parseComparisonExpression function. From here we need to parse the left-hand side and the right-hand side of our expression which continues down the hierarchy of our recursive descent until it hits parsePrimaryExpression which will return an IntegerLiteral as both the left-hand side and right-hand side. Once our comparison expression is created, it is then passed up the hierarchy to the parseVarStatement, finishing up the creation of our variable statement. This statement is the passed back to wherever our parseStatement was called, which will generally be either a function declaration statement or from our program object. Include a UML diagram for parse elements

Section 6: Design trade-offs

We implemented both an interpreter that involved utilizing a tokenizer, parser and evaluate phase of a recursive descent implementation to evaluate and execute Catscript code and a compiler/code generator that compiled Catscript into bytecode that would be executed by the jvm. Everything leading up to the evaluate or compile phase of our code was the same Our implementation of the tree-walk interpreter was fairly simple

Section 7: Software development life cycle model

We used Test Driven Development in developing this project. Test Driven Development is a development cycle that emphasises using tests that target specific functions to ensure the implementation of the code is correct and as bug-free as possible. To achieve this you must craft the code to meet your specifications of functionality and then craft tests to test specific aspects of that code, in our case testing the tokenizer, each specific parse function and the evaluate and compile function.

In this way, we first coded our implementation according to the grammar of Catscript that we were given. For the most part, this was fairly simple to do given the framework professor Gross gave us. The tests used in this portion were simple to debug and were very helpful in making sure our iteration was correct.

We then worked on our parser, first starting by parsing our expressions which was the base level of our grammar. The tests that targeted our expressions were very simple to debug. SinCe our expressions are simple too, I am confident that these tests did a good job encompassing any test case that would involve any of our expressions.

Then we crafted our parser for our statements. This ended up being much more difficult as there was less help from our professor and our grammar to guide the implementation. These statements were more difficult to implement as they required more parsing and sanitation work, such as requiring specific tokens in a specific order while ensuring variables and ensuring expressions were of the correct type (such as the For loop requiring the expression to parse to a ListLiteralExpression.) The tests for this portion were more difficult to debug as some of the information would get lost or buried within our Abstract Syntax Tree and I am not confident that all of the bugs were worked out though using only these tests and the ones supplied by my partner.

Once our parser was working we could start work on the validate, evaluate and compile portions of our project. The validation was simple as it was just checking the type of the expressions within other expressions or statements. The tests surrounding this were few and only checked a couple of cases that would throw the required error, though these tests I felt weren't numerous enough to encompass all of the syntax errors that may occur within the Catscript language.

Then came the work on the evaluate method. This was fairly simple to implement as it was simply setting a variable to the value of the evaluated expression, using that variable or returning the value of the expression. I felt the tests mostly did a good job encompassing any use cases, though as I found out later when crafting tests for my partner and working through my partner's tests for me that my implementation of functions was incorrect and did not support having multiple inputs nor did it support boolean outputs. At this point I felt Test Driven Development shouldn't be the only way to verify the functionality of any code at this stage as the evaluate and parsing logic got too complex to know all of the ramifications, bugs and inconsistencies that would show up at this point. As such it would be very difficult to create a testbase that was comprehensive enough to catch these bugs.

The same could be said about the compile phase but the tests were much more difficult to debug. We couldn't get values for anything other than the bytecode instructions we are stringing together so it was difficult to tell where exactly the logic and code were breaking. In addition to this, the documentation for bytecode was difficult to understand and piece together enough to be usable if I could find any information at all. The tests did not help in this case and made this phase of our project more frustrating than anything else, though when the tests did work it they helped me understand how everything should be working behind the

scenes. As I completed more tests, any subsequent test became easier, however they did not become less frustrating to debug.

All in all, I believe Test Driven Development helped give us some kind of goal to work towards and a way to debug and ensure the logic of our implementation functioned properly, though I don't think any amount of tests written by Member 1, Member 2 and our professor would be able to show if our program is fully-realized. Since we didn't have a user-testing phase of our code, I cannot guarantee that the code is free of any bugs or unforeseen quirks that would cause issues down the road, only that most of the major ones have been caught by our tests. As such I would want to use Test Driven Development as a guideline and in conjunction with some other forms of development when available as user testing with many different people specifically is a very important way to verify and fix the functionality of any program.