

Section 1: Program

<https://github.com/Haedt406/csci-468-spring2023-private/blob/main/capstone/portfolio/source.zip>

Section 2: Teamwork

For the teamwork section. I created three new tests for my partner to verify that their Catscript compiler works as it should. This is called Quality Assurance, or QA. I will first go through the tests I created and ran and then the tests my partner created for me to test my compiler as well. I try to go through what is happening during each step and what it can tell us is happening, such as a function statement or a list expression being created and running successfully.

The first test I created was to test an if-else statement in a function. This program takes a print statement, sends a function call with the number nine, and then once in the function, if the variable that was passed in is equal to 1, print 2, otherwise, print 1. Once the if-else statement has been executed in the function, it returns the value we passed into the function and prints the result of that function we originally passed into the print statement. All of the tests were implemented in “CatscriptFunctionArgsAndReturnEvalTest”, and the code for the first test is as follows.

@Test

```
void customFunctionIfStatmentInFunctionWorks() {
    assertEquals("1\n10\n", executeProgram(
        "function foo(y : int) : int {\n" +
            "if(y == 1){ print(2) }\n" +
            "else{ print(1) }\n" +
            "return y + 1\n" +
            "}\n" +
            "print(foo(9))"
    ));
}
```

The second test that I created for my teammate was to test a function with multiple input variables ran through an if-else and a for statement. This takes a function as a string, and then parses the function, as a function definition statement, and then executes that statement. The variables we put in are an integer, known as variable b, and a Boolean, known as variable c. The return type of the function is an integer. Once the variables are passed into the function it first goes through the if-else statement, testing whether variable c is true or false, and then prints accordingly, then goes to the for loop, in which it only runs through one iteration and prints the value of our integer b. The last part is we return our integer b with two added to it.

@Test

```
public void customFunctionWithMultipleVariablesPassedInForAndIfInside() {
    String function = "function foo(b : int, c : bool) : int
    {if(c==true){c=true}else{print(\"works\")}for(x in [b]){ print(x) } return b +2}\n";
    FunctionDefinitionStatement expr = parseStatement(function);
    assertEquals("works\n1\n3\n", executeProgram(function + "print(foo(1,false))"));
    assertNotNull(expr);
    assertEquals("foo", expr.getName());
    assertEquals(2, expr.getParameterCount());
    assertEquals("b", expr.getParameterName(0));
    assertEquals("c", expr.getParameterName(1));
    assertEquals(CatscriptType.INT, expr.getParameterType(0));
    assertEquals(CatscriptType.BOOLEAN, expr.getParameterType(1));
}
```

```
}
```

These previous two tests show us that we are able to successfully tokenize, parse, and then execute a function, if-else, for, and print statement; with our function statement being assigned the name foo. This also determines that a if-else statement can properly handle conditions, that are implemented as a list of statements, and a body of set expressions. The last piece shows a successful parse return statement that can return an additive expression, and then finally print out the value returned from the function.

For our third test, I tested to see if a for statement could handle a list expression of integers, that would iterate through the number of elements in the list, and then use an if-else statement to determine if it was iterating through it in the proper order as well as proper amount of iterations. We send in a list literal expression with three integers in it and are able to successfully print three times based on the conditions of the if-else statement.

```
@Test
```

```
public void customForWithIfElse() {  
    assertEquals("false\ntrue\nfalse\n", executeProgram("for(x in [1,4,7]){ if(x  
==4){print(\"true\")}else{print(\"false\")} }"));  
}
```

Tests my partner created for me to run QA on.

My partner provided three tests for me to run. I was able to pass all three of his tests. Down below are the tests. I will also describe how they ensure parts of my compiler are working successfully as I did with the tests I created.

The first test my partner created was a test that would create a variable and assign it a value of three. Then we take the variable, x, and run it through a series of if statements, with the last one being an if-else statement. When the variable is created and assigns it the value of 3, it successfully determines that 3 is an integer. Next, we go through the if statements and if x is equal to the specified variable to test against, it will print out that variable. Since we have multiple if statements, it successfully checks on the third one that x == 3 and then prints 3. This tells us that our tokenizer was successful in scanning, then we were able to create a variable statement as well as a series of if statements, and it followed it in successive order. Since we have multiple if statements, where if a condition isn't met it doesn't go into that if statement to perform its designated functions, it continues on and keeps running. We know that it does this successfully since our last if statement is an if-else statement that successfully determines that since our variable x is not == 4, it has the else condition where it prints out the value 5.

```
@Test
```

```
void longLongIfStatementWorks() {  
    Assertions.assertEquals("3\n5\n", this.executeProgram("var x = 3 \nif(x == 1) { print(1) }\nif  
(x == 2) { print(2) }\nif (x == 3) { print(3) }\nif (x == 4) { print(4) }\nelse { print(5)}"));  
}
```

The second test to look at takes a for loop and runs through the loop the proper number of times and then using if-else statements determines if the number being tested, in this case x, which is in a list, is equal to a value, it prints "true", if the condition is not met, it prints "false". This test determines that we can successfully tokenize the syntax, then create a for statement, take a list, and iterate through the list, assigning the value of x to each variable in the list as we iterate through the list. Since in the for statement, we are testing to see whether the value of x is equal to 3, it iterates

through 3 times due to there being three digits in the list, and then determines that the first two values are not equal to 3 in our if-else statement, and prints “false” twice, then on the last iteration, since the value of x is assigned to the integer 3, the if statement properly determines that the condition for the if statement has been met and prints “true”. This shows us that we can successfully create list expressions, for statements, and if-else statements.

```
@Test
public void personalForIfLoop() {
    Assertions.assertEquals("false\nfalse\ntrue\n", this.executeProgram("for(x in [1,2,3]) {\nif(x == 3){ print(\"true\") }\nelse { print(\"false\") } }"));
}
```

The final test created by my partner created assigns the Boolean value true to “hold”, and then checks with a if statement if the variable “hold” is equal to true, then go into the if statement and iterate through a for loop. This test shows us that a variables type can be successfully determined without having to define its type, meaning that it is a dynamically typed programming language when we check to see, using an if statement, if the value “hold” is equal to true. Once inside the if statement, it has a for loop to iterate through, since there are 8 values in the for loop, it iterates through 8 times, in the for loop, it has an if statement where it tests if the value “x” is equal to 4, if not then it goes onto the next part which is an if-else statement, where if the value of “hold” is equal to true, then print true. In the for loop, we are iterating through a list expression and assigning it to x based on its position in the for statement. This successfully shows us that, again, a list expression is being created properly and we assign the variable, in this case “x” to the number that it is at in its iteration of the for loop. Since we are iterating through the loop 8 times, when x is equal to 4, it prints 4 and then goes onto the next if-else statement, and since the conditions of the if-else statement are always true since it is checking the value of our variable “hold”, it will always print true. Like the previous tests, it tells us that we have properly implemented an if and if-else statement, a for statement, list expressions, and variable statements as to the specifications of the Catscript language.

```
@Test
void personalIfForIfOutput() {
    Assertions.assertEquals("true\ntrue\ntrue\n4\ntrue\ntrue\ntrue\ntrue\ntrue\n",
this.executeProgram("var hold = true \nif(hold == true) { for(x in [1,2,3,4,5,6,7,8]) {\nif(x == 4) {
print(4) } if (hold == true) { print(\"true\") } else { print(\"other\") } } }\nelse {
print(\"failedHold\") }"));
}
```

Our team worked on the capstone project by creating three tests for each other to test our compiler on for QA and also writing each other documentation on Catscript language.

For the teamwork portion of the project, I, Benjamin Haedt, spent about 15 hours writing 3 tests for my partner, testing my partner Jace’s tests, and finally writing my Catscript documentation.

Section 3: Design pattern

The design pattern that I will explain why we used it instead of coding it directly is located in CatscriptType as a function called getListType.

The original code was as follows.

```
public static CatscriptType getListType(CatscriptType type) {
    return new ListType(type);
}
```

While the above code works, it is not efficient since every time we call getListType we new up a list.

So, for example, if we call `getListType` ten different times with integer then it will create ten different instances of the list type. This will take up more memory and in an inefficient way of checking the list type. We can optimize the code by creating a map that will hold the list type for us, so once we create a list, it will check the map and if it sees that the map already has the lists list type in the map. If it does, then it will return the list type without having to create multiple instances of the list to check for its type. Essentially, it is caching the list type of a list, so when we want to check the list type later, it will already be available to us, and we will not have to create new instances of the list.

```
public 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. Include the technical document that accompanied your capstone project.

NOTE The Catscript Guide my parter provided me with has compatibility issues with my documentation since I used word to create my documentation and he used markdown. So, I will include a link to his documentation and how he had it formatted. The documentation can also be found in the /capstone/portfolio folder.

[https://github.com/Haedt406/csci-468-spring2023-private/blob/main/capstone/portfolio/Catscript Documentation%20from%20teammate%20for%20Task%204.pdf](https://github.com/Haedt406/csci-468-spring2023-private/blob/main/capstone/portfolio/Catscript%20Documentation%20from%20teammate%20for%20Task%204.pdf)

Catscript Guide

The following documentation addresses the Catscript programming language and the syntax necessary to code in the functional language. The Catscript compiler is based on a Java transpiled system that outputs through the JVM Bytecode systems, but is centralized to use just Catscript coding components listed below.

Introduction

Catscript is a simply functional language that works with functions, expressions, statemnets, variables, and other syntax components familiar to Java, Python, and other functional components in most languages. The following is an example of the base look and coding of the Catscript language:

```

var x = "foo"
print(x)

if(x=="foo")
{
    print("truth")
}
else
{
    print("false")
}

```

The language is statically and strongly typed, allowing us to mimic some of the basic type functionality of Javascript and Python languages. The following documentation will address the usage of expressions, statements, functions, types and variables, and the syntax necessary to produce error-free code with Catscript.

Example of the Catscript Language

- ♦ Examples: Expressions and Statements

```

var x : int = 10
var arithVar : int = (x + 1) / (4 * x)
var y : string = "hello"
var listDemo : list<int> = [1, 4, 5, 6, 7]

var z = true // Assumes the Type
var nullType : object = null

print(x) // Prints Variable
print(y)

if (z == true) // Determines if Z equals true, boolean
equalities.
{
    z = false
}

```

```

}
else if (arithVar >= 1)    // Compares arithmVar to see if it is greater
                           than or equal to one, comparisons.
{
    print(nullType)
}
else
{
    print(y)
}

for(iterator in [1, 2, 3, 4])    // Iterates over a list of 4 values,
                                prints the values and parts of our list, listDemo.
{
    print(iterator)

    print(listDemo[iterator])
}

```

- ◆ Examples: Functions

```

function foo() {                                // function foo() executes a
printing of added variables

    var foo_x : int = 0
    var foo_y : int = 10

    var foo_z = 0 + 10

    print(foo_z)

}

function bar(a: int, b: int) : bool {           // function bar() returns a
boolean, decided by checking if a and b equal 10

    var output : bool = null
    var total : int = (a + b)

    if (total == 10)
    {
        output = true
    }
    else
    {
        output = false
    }

    return output
}

print(foo())    // Will Print 10

```

```
print(bar(5, 5))    // Will Print TRUE
```

Programming in Catscript

- ♦ Types and Variables

Types

The following topic covers the types that are usable within the Catscript language, which will hold similar typing as most common languages. The types used in Catscript are as follows: **INT**, **STRING**, **BOOL**, **LIST**, **NULL**, and **OBJECT**. The types are used to define *variables* and *functions* that will be used within the language. Types will be demonstrated below in several examples, the statements, variables, and functions will be defined further on.

```
expression - statement NO_TYPE ;
expression - statement : int ... ;
expression - statement : string ... ;
expression - statement : bool ... ;
expression - statement : object ... ;
expression - statement : list [... : TYPE ] ... ;

expression - statement : ANY_TYPE -> set to NULL ;

function "name" ( ... ) : TYPE { ... } ;
```

Variables

The following topic covers the variables in Catscript and how they are created to be used in functions, statements, and with expressions. The variables will have types and will be demonstrated in the examples below.

```
var value = TYPE_VALUE ;
var value : TYPE = TYPE_VALUE ;

var number : int = 0 ;
var word : string = "string word" ;
var truthy : bool = true ;
var listy : list<TYPE> = [val1, val2, val3, ...]
var random : object = VALUE ;

var nullEx = null ;
var nulledType : TYPE = null ;
```

- ♦ Expressions

Additive, Subtractive, Multiplied, Divided

The following examples demonstrate the different forms of arithmetic capabilities of the Catscript language. The language is capable of adding, multiplying, subtracting, and dividing integers. Additionally, Catscript can concat strings, nulls, and some objects with the right typing. We also see how parenthesis are capable of bring priority to arithmetic values and equations. The following information is demonstrated in the examples below.

```
val1 + val2, var total = val1 + val2, var total : STRING_INT = (val1 + val2)
OR val1 + val2 [TYPES MUST MATCH VAR TYPE]
val1 - val2, var total = val1 - val2, var total : INT = (val1 - val2) OR
val1 - val2
val1 * val2, var total = val1 * val2, var total : INT = (val1 * val2) OR
val1 * val2
val1 / val2, var total = val1 / val2, var total : INT = (val1 / val2) OR
val1 / val2

EX
var endString : string = "word"
var number_value : int = 1

var string_concat : string = "String " + endString ->      "String word"
var string_int_cat : string = "Value: " + number_value ->   "Value: 1"
var string_null_cat : string = endString + null ->          "wordnull"

var int_add = 10 + 100 ->      110
var int_minus = 5 - number_value ->  4
var int_multi = 10 * 10 ->      100
var int_divis = 50 / number_value ->  50

var long_arithm = 5 + 10 * (number_value / 5) ->  "3 = 15 / 5"
```

Boolean, Null

The following example shows the use case of Boolean and Null values. The use of Booleans are often used in choices and systems of checking within a function or output. The use of Nulls can be used to return certain outputs in failed cases and is often used as a placeholder for systems. The following information is demonstrated in the examples below.

```
true, false, var bool_value = true, var bool_value = false

null, var null_value = null, var null_value : ANY_TYPE = null

EX
var big_value : int = null      // Can be used to predefine values before we
use them, so we do not accidentally trigger choices
var val : bool = true          // Set variables to a boolean value when
their TYPE is bool
```



```

var ex_val = false                // Assumes the boolean type value

if (val == null)                  // NULLS can be used as checks in choice structure
and error structure
{
    ... body ...                  // We could return a function if our value is NULL,
to protect the program
}

if (val == true)                  // checks if a variable or literal is equal to the
boolean true
{
    ... body ...
}

if (val == false)                /// checks if a variable or literal is equal to the
boolean false
{
    ... body ...
}

```

Comparisons, Equivalence

The following example shows the use case of Comparisons and Equivalence in the Catscript language. Comparisons are essential to making logical pathed descisions in the Catscript compiler environment, while Equivalence is how most of our choice structure works in the language. We can see below how equivalence allows us to make coordinate choices and how it can be used alongside comparisons to become even more specific with our choice structures.

```

val1 == val2, val1 != val2

val1 < val2, val1 <= val2
val1 > val2, val1 >= val2

EX
var val1 : int = 1
var val2 : int = 5
var bool_val : bool = true

if (bool_val == false)           // IF values are equal then its true
{
    ... true body ...
}

if (bool_val != false)           // IF values are not equal then its true
{
    ... true body ...
}

```

```

if (val1 >= val2)           // IF values are greater than or equal to second
values then its true
{
    ... true body ...
}

if (val1 < val2)           // IF values are less than to second values then
its true
{
    ... true body ...
}

```

- ♦ Statements

If Else Statements

The following example shows the use of If Statements and how to make proper choices with variables, literal type values, and other systems to build complex deciding structures. This system of statements allow an individual to code choices into their program, below will show examples of the following statements.

```

if (expression) {
    ... true body ...
}
else {
    ... false body ...
}

if (expression) {
    ... 1st true body ...
}
else if (expression) {
    ... 2nd true body ...
}
else if ... {
    ... nth true body ...
}...
else {
    ... false body ...
}

EX
if (val1 == val2)
{
    print(val1)           // Prints val1 if values are equal, and val2 if they
are not.
}
else

```

```

{
    print(val2)
}

if (x == 1)          // Switch-like system of executing body expressions
when conditions are true in the if.
{
    print(x)
    bool_value = true
}
else if (x > 5)
{
    print("Bigger")
}
else if (x > 100)
{
    print(100000)
    x = 100000
}
else
{
    print("poor")
    x = 0
    bool_value = false
}

```

For Statements

The following example shows the use of For Statements and how to make looping systems with our Catscript data structures. The system will use variable structures to help iterator over the loop, which can use global and local scoped items.

```

for (variable in [val1, val2, val3, ...]) {
    ... body ...
}

EX
for ( x in [1, 2, 3] ) {           // Iterate X over the literal list, [1,
2, 3]

    print(x)

    if(x == 1)
    {
        print("Start")           // Print structures can take literal
values
    }
}

```

Functions Header

The following emphasizes the header and how it is used to initialize a function in Catscript. The function header is necessary for building the main structure identified by our Catscript compiler.

```
function funcName (...) {... body ...}  
  
EX  
function fooBar (...) {... body ...}
```

Functions Parameters and Variables

The following emphasizes the parameters and variables that are implemented into the header of the function. The parameters and variables must be defined, along with the function type, to use variables that are necessary to our functions execution. Not all functions will have the parameters filled in, or the type defined.

```
function funcHeader (parmA : TYPE, paramB : TYPE) : Func_TYPE {... body ...}  
  
EX  
function fooBar (value_A : int, value_X : bool) : int {... body ...}  
  
function fooBar (NO PARAM) : NO_TYPE {... body ...}
```

Functions Body

The following emphasizes the function body data structure, which holds the specific processing and combination of data structures within Catscript that an individual can use. The body will return values, using a return statement, and output other values if not return is specified.

Below the example will showcase all of the following components of a Function and its internal data structures.

```
function funcHead (parmA : TYPE, paramB : TYPE) : Func_TYPE {  
  
    body_variables OR body_expressions  
  
    body_statements  
  
    return_statement --> REQUIRED IF TYPE DEFINED  
}  
  
function funcHead (NO PARAM) : NO_TYPE {  
  
    body_variables OR body_expressions
```

```

body_statements

return_statement --> OPTIONAL IF TYPE NOT DEFINED
}

EX
function helper (key : int, value : string) : bool {

    var key_verified : bool = null

    if (key == 1)                                // Key System to Verify Value state
    {
        value = "output is 1"
        key_verified = true
    }
    else if (key == 2)
    {
        value = "output is 2"
        key_verified = true
    }
    else if (key == 3)
    {
        value = "output is 3"
        key_verified = true
    }
    else
    {
        key_verified = false
    }

    return key_verified                          // Returns Checked State
}

function execution (adder : int) {

    var multiply_Key : int = 5
    var new_value = ( adder * multiply_Key)      // Type is Assumed

    return new_value
}

function cat (newEnd : string) {

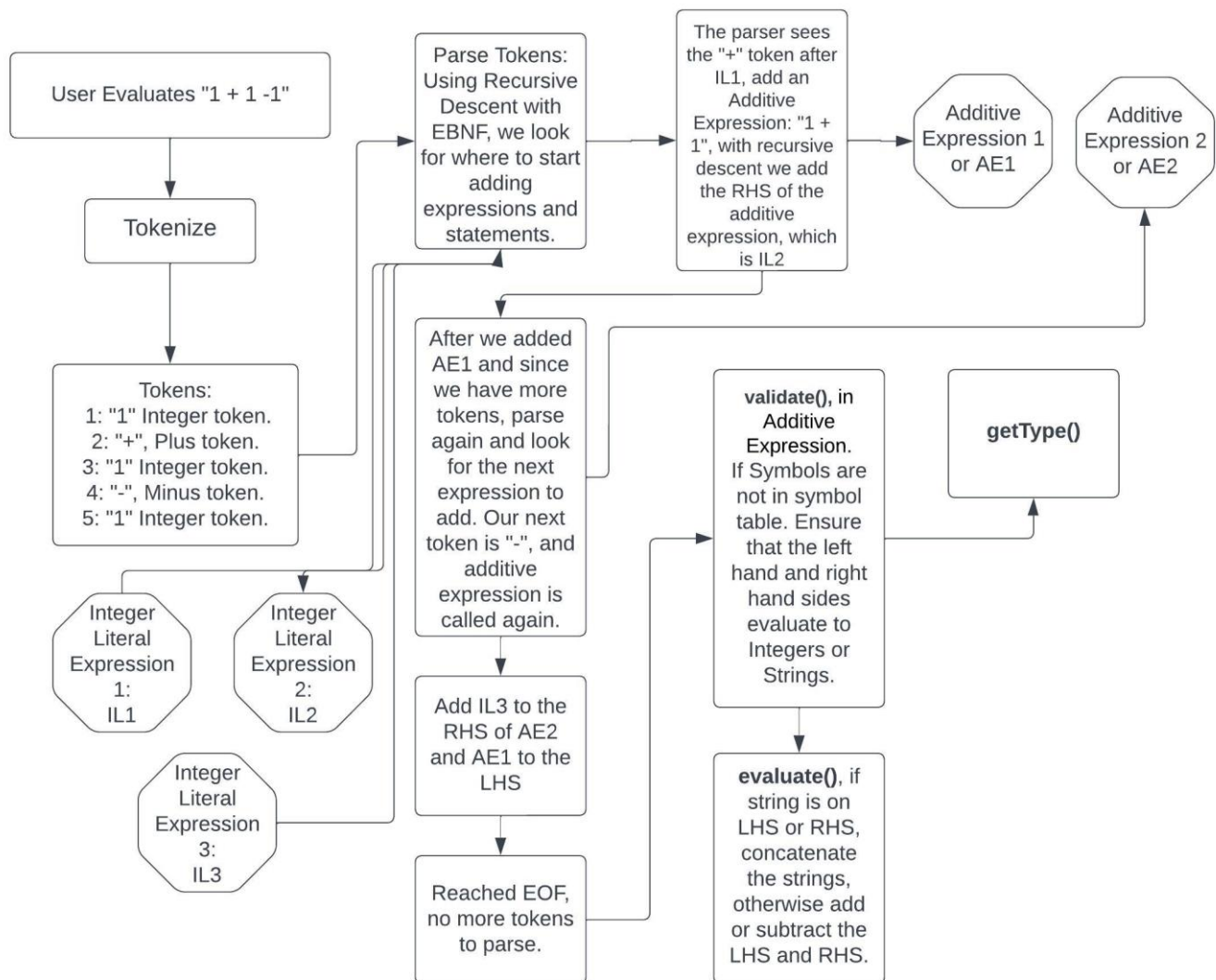
    var phrase : string = "Hello, my name is "
    var new_phrase : string = phrase + newEnd    // Catscript
    concantentation of Strings

    print(new_phrase)                           // No Return
    is Needed, The function just executes when called.
}

```

```
        cat("dog")                // Takes a string and prints out a
        phrase,
    execution function.
    print(execution(1))          // Function returns a value that can be set
    variable or
    var checker : bool = helper(1, stringValue)    // Function returns
    boolean
```

Section 5: UML.



Section 6: Design trade-offs

For our compiler we created a compiler based on recursive descent. We chose recursive descent for the language we are parsing for a few reasons. To start, it is considered easier to implement than other parsing techniques, such as LR parsing, because it directly maps to the grammar of the language being parsed. Recursive descent parsers can also make it quicker to develop and to debug the compiler. And the last benefit I will mention is that they can be easier to read and understand. Looking at the flipside of this, recursive descent parsers can be harder to implement and scale to larger grammars. As the grammar grows in complexity, the number of recursive function calls increases as well, this could lead to stack overflow errors and a slower performance. In our language, we used Extended BNF, which recursive descent parsing can handle, but attempting to implement left-recursive grammars could have presented challenges for us.

Section 7: Software development life cycle model

The model that we used to develop our capstone project was Test Driven Development. Test driven development is a methodology used in software development in which tests are written for code before the code is written. In our project, we wrote tests on how our compiler should work and then we ran those tests to see if it worked as expected. If the test failed, then we would go back and adjust the code. This helped us develop the Catscript compiler because it allowed us to visualize the

language being written and used before we actually had a functioning compiler.

Another advantage of using test-driven development is that it can help to improve the overall quality of the code. By writing tests before writing code, developers are forced to think about the requirements and design of the software at a more detailed level. This can lead to more modular and reusable code, as well as better documentation and understanding of the codebase. In addition, this model of development for code can also help reduce the risk of bugs and errors in the codebase.