Teamwork

When approaching this project, team member 1 first assessed the needs of the project and wrote the first draft of the code. Team member 1 then gave this code to team member 2 who reviewed it and supplied team member 1 with detailed documentation about the code. Once team member 1 received team member 2's documentation, team member 1 reviewed it, team member 1 and team member 2 discussed how to make it better, and we both suggested changes that would improve the documentation. Throughout this process, the ability to trade ideas and discuss changes was vital to the improvement of the code. Overall, team member 1 spent about 70% of the project time on writing, testing, and improving the code while team member 2 spent about 30% of the project time creating documentation and reviewing errors.

Team member 2 also provided team member 1 with tests to thoroughly check the major aspects of the code team member 1 had written. The first test team member 2 provided team member 1 with was meant to check if variable assignments within functions work as intended. The next test we ran was to check that, if the parameter had the wrong type, the compiler would throw an error. The final test we ran was to make sure that nested for loops within other for loops execute properly. After running all of these tests, team member 1 was able to find and fix faulty aspects of the code. Thanks to these tests, we were able to solidify the validity of our code. Team member 2's tests are below.

```
@Test
void variablesInsideFunctionsAssignProperly() {
    assertEquals("10\n", executeProgram("var x = 1\n" +"function foo() {\n" +
        "x = 10" +
        "}\n"+
        "foo()\n"+
        "print(x)"));
}
@Test
void parametersOfWrongTypeThrowsError() {
    assertEquals(ErrorType.INCOMPATIBLE_TYPES, getParseError("var x = \"asdf\"\n" +"function foo(y : int) {\n" +
        "print(y)" +
        "}\n"+
        "foo(x)\n"));
}
@Test
void doubleForLoopsExecutes() {
    assertEquals("3\n9\n", executeProgram("var x = 0\n" + "var y = 0\n" + "function foo() {\n" +
        "for(i in [1, 2, 3]) { x = x + 1\n " +
        "for(l in [1, 2, 3]) { y= y + 1 } }" +
        "}\n" +
        "foo()\n" +
        "print(x)\n" +
        "print(y)\n"));
}
```

Design Pattens Used

Memoization is a pattern that is useful no matter if one is writing procedural or object oriented programming. This pattern uses a hash map to store values from expensive function calls and look them up later. If the function takes exponential time, the function will only ever need to calculate the value once. Once calculated the value can be looked up in constant time. These values are only available during runtime, but even so, it still will save precious time. If we were to code this directly the function would have to calculate each value every time we use that value which would take exponentially more time.

One such function call within Catscript that is especially costly is the getListType method. This can be seen in the class under src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java. Creating new list types is an incredibly expensive operation so by using this pattern we only ever have to create them once. If we were to code this directly on the other hand it would take much more time to get these values. The code involved for this method has been extracted from the source and is shown below.

```
private static final HashMap<CatscriptType, ListType> LIST_TYPE_CACHE = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    if(LIST_TYPE_CACHE.containsKey(type)) {
        return LIST_TYPE_CACHE.get(type);
    } else {
        ListType listType = new ListType(type);
        LIST_TYPE_CACHE.put(type, listType);

        return listType;
    }
}
```

CatScript Documentation

General Information: CatScript is statically typed programming language with a simple type system, that has 6 types, int (32 bit Integer), string (a set of characters), bool(true/false), list<x>(a list of type x), null, and object(any type of value). The types of all variables and functions are known at compile time. CatScript has built-in functionality such as variables, variable assignments, print statements, if statements, for loops, simple

mathematical expressions, comparison expressions, equality expressions, and functions.

Types:

int - This is a 32-bit integer that can store any number between the values 2,147,483,647 and -2,147,483,648.

bool - This type is a Boolean value, and variables of this type can be set to true or false.

list - A list can be used to store a collection of other types. But lists can not be modified after being created.

null - This is just a null value indicating the absence of a value in a variable.

string - This type is a collection of characters such as “random string” or “abc123”.

object - This type can be assigned the other types. An example would be Object obj = “random value”. That Object obj now stores a string value. You could also say Object obj2 = 123. Object obj2 stores an integer value of 123. Objects cannot be assigned to other types. You can not do String str = obj.

Functionality:

Variables – Variables can be declared using either of the following formats. First, you use the keyword Var followed by the identifier/name and then an optional semicolon with a type and then an equal sign followed with the value you want to be assigned to the variable.

var x = 10 Or var y : list<int> = [1,2,3]

The first format assumes the variables type based on the value provided, the second declaration method has an explicit type where you tell it what the type is. The variables in cat script are stored and can be called later in the same scope to use the assigned value.

Assignment – You can also assign values to variables using the following format.

x = 20

This allows you to change the value that a given variable points to. A couple of important things to note with assignment variables is that the type of the expression that you are assigning to the variable needs to match with the variable’s expected type. The variable also needs to be declared in the scope in which you are assigning the value.

Unary Expressions – Unary expressions are used to get the inverse of whatever follows. To use a unary expression, you can put a – before a number to get the negative number. You can also use the Not keyword to get the inverse of a boolean. The Not can also be used before an expression of function that produces a bool to inverse the output. Below are a few examples of unary expressions.

-1 or -4

not true of not (10<9)

Parenthesized Expressions – Parenthesized expressions are used to enforce a priority in the execution of the stuff inside of the parentheses. It can be used to enforce correct mathematical execution or to apply expressions to other expressions as shown below.

7 * (8 + 9)

not (8<9)

Factor Expressions – Factor expressions are used by using the multiplication symbol or the division symbol between two integers, as shown below.

10 * 7 OR 90/10

The first example would result in 70 or 10 multiplied by 7 and the second example would result in 9 or 90 divided by 10.

Identifier Expression – Identifier expressions represent the names assigned to variables or functions that are declared in a program. Below is an example where variableName is the identifier.

var variableName : string = “var”

Additive Expression – Additive expressions use the addition and subtraction sign. The addition symbol can be used to add two integers together or can also be used to concatenate a string to another value. Below is an example of mathematical addition usage

9 + 8

This would result in a value of 17, you can also use the addition operator to concatenate a string to another value as shown below.

“string “ + 123

This would result in the String value of “string 123”. The subtraction operator is only use to subtract the second integer from the first as shown below.

17 – 9

This would result in the integer 8.

Print Statements – In Catscript you can print values out to the console using print. This built-in functionality allows you to print out the string value of what is in the parentheses following the print keyword. Below are a couple of examples of the use of print in catscript.

print(1)

print(“this string value”)

Comparisons Expressions – In CatScript there is built in comparison and equality tools that evaluates to true or false depending on the values used. Using comparison expressions allows two integers to be compared and produce a boolean result. Built in comparisons are greater than, less than, less than or equal to and greater than or equal to. Shown below is how to use each comparison.

Greater Than - 8 > 9 - Evaluates to true

Less Than - 9 < 8 - Evaluates to false

Greater Than or Equal to - 8 >= 9 - Evaluates to false

Less Than or Equal to - 8 <= 9 - Evaluates to true

Equality Expressions – For equality expressions, you can compare values of different or the same type to check if two values are equal or not equal. Shown below is how to use equals and not equals.

Equal - 8 == true - Evaluates to false

Not Equal - 8 != 9 - Evaluates to true

If Statements – CatScript has built-in if statements which allows you to have conditional execution. You provide an expression that will be evaluated as true or false. If the expression evaluates to true, then the Statements within the if are executed other wise they are not executed. The implementation of the if is shown below.

if(x = 10){statements}

After an if statement you can optionally use an else which will execute the statements in the else if the expression evaluates to false. The implementation of if-else is shown below.

if(x == 10){statements}

else{statements}

For Loops – Built in for loops allow you to iterate over every value in a list and execute a set of statements for every value in a list. When using a for loop there is a temporary variable that while in the for loop you can use that variable to access the current value from the list, in the example provided the x points to the current value in the list. The example below shows how to use a for loop.

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

Something to note with for loops is while in for loop you have access to the variables that are available in the scope where the for was used, but within a for loop you can declare a new variable that can only be used within that loop and the new variable will not be usable outside of the loop.

Functions – Functions must be declared and then called. In CatScript a function declaration must start with the function keyword and then be followed by what you want to name the function. After the name, there must be a set of parentheses where you put all of the needed argument declarations. After the parentheses there is the option to put a semicolon followed by a return type, if you put a return type here the function must have a return statement to return something of the specified type. Then there are the brackets that will contain all of the statements/ functionality for the function. Below is an example of a function declaration.

function foo(x) { print x }

Similar to in for loop, while inside of a function you can use variables that are declared in the same scope as the function, while also being able to declare new variables within the function that will be unusable after the function finishes execution. So if you want to keep values calculated in a function they should be returned.

Function Calls – To call a function you need to use the function name followed by parentheses that contain the values that you want to pass into the function. It is important to note that the values you put in the parentheses must match the order and type, if specified, of arguments that are in the function declaration. This is useful for running the same code on multiple different values and avoids having multiple instances of the same code. Below is a function call for the function declared above.

foo(“this is the value”)

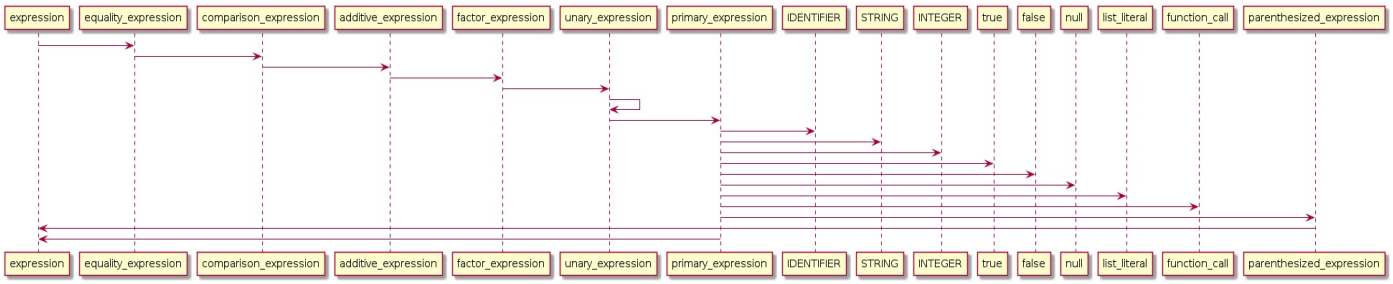
Return Statements: Return statements are used at the end of a function to end the function and return a value if specified. If a function does not have a return parameter specified, a return can still be used in an if to end the function and not finish the rest of the statements. A return is also used to specify what value to return if the function has a return type specified in the declaration. Below is an example of a function with a return.

Declaration: function foo(x) : string { return “this is “ + x }

Function Call : string str = foo(“a sentence.”)

The function call sets str to be “this is a sentence” because the function returned the string that was concatenated.

UML



This diagram shows the expressions in a recursive descent compiler. Recursive descent follows the idea that everything is connected recursively. Each expression comes from expression and is connected to each other and ultimately leads back to expression. In this way no matter if it's an additive_expression or even a unary_expression every expression is in one tree. At the root node of the tree there will always be the operator performed first. At each leaf node there will be either an identifier, string, integer, true, false, list_literal, or function_call. These are the operands. In all the parent nodes there will be other operators. This also means that the order in which expressions are calculated is enforced by how far down in the tree they are. Recursive descent ultimately creates a tree of values where the operators near the top of the tree are least important, calculated first, and the operators at the bottom are most important, calculated last. If a parenthesized expression is used the parenthesized expressions will always be done first so the expression used within it will be moved higher up in the tree.

In this diagram identifier, string, integer, true, false, null, list literal, and function call are all potential operands for operators. Even though it's possible to parse a list literal in an additive expression, in such a case we would have an error. This leads us to a problem that is common within recursive descent algorithms. When a recursive descent algorithm finds an error it will often just keep reporting new errors until the end of the file, even if they don't exist. This shortcoming can be fixed pretty easily, but it brings into perspective another interesting side effect of every expression being connected.

Design Tradeoffs

In the world of compilers there are two major contenders, recursive descent and code generators. Recursive descent is both easier to write and easier to debug. In recursive descent everything is connected to each other recursively. Expressions can be chained together and statements can be passed to each other with no extra code written because of its recursive nature. Another benefit to recursive descent is that it is used in the professional world. Because of its recursive nature, recursive descent is also often a lot simpler than code generators. This is one of the arguably easiest ways to write a parser by hand. This is because when writing a recursive descent parser we have control of how every token is used.

Code generators on the other hand, require regular language to be written. Regular languages often have cryptic syntaxes and the same symbols often have different meanings. Another downside to code generators, is the code produced is often very messy. This appears in the form of very deeply nested if statements and cryptic variable names. Another reason parser generators are often not a very good idea is that code that is generated should not be edited. This is because if the code is regenerated the changes will be overwritten and all that work will be lost. One reason that parser generators are popular is regular languages are very concise and can be written very fast. The regular language file written is also often a lot smaller with the downside of the compiler being a lot more lines. In this way a compiler can be written in a vastly shorter amount of time if you know regular languages very well. This is most likely the reason parser generators are popular among academics.

Therefore, it's a question of whether we want to write a clean compiler by hand from the ground up, or learn how regular languages work and generate one ourselves. That decision is one that should be left up to the compiler writer. But, whichever path we take in the end, we have learned something useful. We will end up with a compiler in the end that compiles a language that we made up. That is, in my opinion, one of the most amazing parts about coding.

Software Development Cycle

Test driven development is a technique that involves creating tests for different areas of the program. When running tests, the first result one receives will either be a pass or fail. Most of the time the test will initially fail, but once the problem is fixed, the test can be rerun and will update the result. If everything has been fixed correctly, this test will pass with no problems. In this way each aspect in the code can be tested individually with a specific error message for each aspect.

In Catscript we used a large number of tests to check the functionality and validity of our project. These tests ranged from parsing tests to execution tests and even to bytecode tests. The amount and variety of tests does a good job of showing just how versatile and powerful test driven development can be. As you can imagine, the implementation of this development strategy was an essential aspect when it came to debugging the code for our project.

Without utilizing test driven development, the process of developing this project would have been much more difficult. If we hadn't utilized this development system, we would have spent many unnecessary hours going back to our code and trying to find and fix all of the problems. The tests also helped our team to have an efficient and streamlined direction for our communication when it came to the problems within the code. If a different development method had been used, we don't believe our time and energy would have been used as effectively.