Section 1: Program

Program included zipped.

Section 2: Teamwork

Our teams for this course consisted of teams of 2 (3 in special circumstances). For the course we would develop our recursive descent parser compiler on an individual basis. The compiler would then be documented by our partners explaining the features of the program. My compiler was also tested by my partner for proper parsing and functioning. We would then review each other's documentation and make necessary corrections as well as run the tests and make the necessary corrections as well. The percentage of time spent on the project vs ourselves was an even spilt as we were checking and polishing each other's work. The tests used to test my code from my partner can be found below. The documentation for my compiler made by my partner can be found in section 4. Only minor bugs were found and were fixed same hour, my tests developed for my partner can be found in my source code in src/test/java/demo/PartnerTest.java.

```
@Test
public void stringConcatTest() {
// Testing string concatination
assertEquals("this is testing string concatination", evaluateExpression("\"this is testing\" + \" string conca
// testing integers turn into a string value
assertEquals("string concatination with integers1", evaluateExpression("\"string concatination with integers\'
// checking if null value with string concat turns null value into string "null"
assertEquals("null is a string value", evaluateExpression("null + \" is a string value\""));
// testing all three together
assertEquals("string value null plus string value 1", evaluateExpression("\"string value \" + null + \" plus s
}


@Test
public void comparisonWithFactorExpression() {
    // with multiplication
    assertEquals(false, evaluateExpression("1 * 2 > 2"));
    // with parenthesized multiplication
    assertEquals(false, evaluateExpression("1 * (1*2) >= 3"));
    // with division
    assertEquals(true, evaluateExpression("10 / 5 <= 2"));
    // with parenthesized division
    assertEquals(true, evaluateExpression("10 / (10 / 5) < 6"));
    // with multiplication and division
    assertEquals(true, evaluateExpression("10 * 10 / 10 > 9"));
    // with parenthesized multiplication and dvision
    assertEquals(false, evaluateExpression("10 * 10 / (5*2) / 10 * (10 / 2) > 6"));
}

@Test
public void ifStatementWorks() {
    // testing basic if statement
    assertEquals("true\n", executeProgram("if(10 < 11){ print(true) }"));
```

```
        // testing false if statement with no else statement
        assertEquals("", executeProgram("if(10 > 11){ print(1) }"));
        // testing true statement with else
        assertEquals("true\n", executeProgram("if(10 < 11){ print(true) } else { print(false) }"));
        // testing false if with else
        assertEquals("false\n", executeProgram("if(10 > 11){ print(true) }  else { print(false) }"));
        // testing false if, true if else
        assertEquals("false\n", executeProgram("if(10 > 11){ print(true)} else if(10 < 11) {print(false)}"));

    }
```

Section 3: Design pattern

One design pattern amongst many that is used in this project is the memoization call. This pattern is useful because it allows the program to recall previously computed computations. Now we could have just coded directly but this pattern allows for the best utilization of resources which is the biggest basis of dynamic programming and caching. This is important because of where compilers sit on the development stack. Since most things are built off of the compiler it is of upmost importance to lower the overhead as best we can and be as efficient as possible. The simple nature of memoization makes it a low-cost high reward tool. It was a no-brainer to approach the problem of computation with memoization but with that being said if you had no knowledge or were not trying to make the code as efficient as possible it is more than likely something like this could have been overlooked.

Now this code sits in the CatScript Type file which sits in our parser. This code is used when we call CatScript Type which is used to determine the type of variable we are dealing with in the parser this could be called many times especially if dealing with large amounts of lists where the computation of component types can be resource consuming the multi list types also add to this and additional dimensions in particular can be most consuming.

Memoization Call (Specific Code)

```
    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, cache.get(type));
            return cache.get(type);
        }
    }
```

CatScriptType.java (Where memoization is used)

```
package edu.montana.csci.csci468.parser;

import java.util.HashMap;
import java.util.List;
import java.util.Objects;
```

```
public class CatscriptType {

    public static final CatscriptType INT = new CatscriptType("int", Integer.class);
    public static final CatscriptType STRING = new CatscriptType("string", String.class);
    public static final CatscriptType BOOLEAN = new CatscriptType("bool", Boolean.class);
    public static final CatscriptType OBJECT = new CatscriptType("object", Object.class);
    public static final CatscriptType NULL = new CatscriptType("null", Object.class);
    public static final CatscriptType VOID = new CatscriptType("void", Object.class);

    private final String name;
    private final Class javaClass;

    public CatscriptType(String name, Class javaClass) {
        this.name = name;
        this.javaClass = javaClass;
    }

    public boolean isAssignableFrom(CatscriptType type) {
        if (type == VOID) {
            return false;
        } else if (type == NULL) {
            return true;
        } else if (this.javaClass.isAssignableFrom(type.javaClass)) {
            return true;
        }
        return false;
    }

    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, cache.get(type));
            return cache.get(type);
        }
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        CatscriptType that = (CatscriptType) o;
        return Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    public Class getJavaType() {
        return javaClass;
```

```
    }

    public static class ListType extends CatscriptType {
        private final CatscriptType componentType;
        public ListType(CatscriptType componentType) {
            super("list<" + componentType.toString() + ">", List.class);
            this.componentType = componentType;
        }

        @Override
        public boolean isAssignableFrom(CatscriptType type) {
            if (type == NULL) {
                return true;
            } else if (type instanceof ListType) {
                ListType otherList = (ListType) type;
                return this.componentType.isAssignableFrom(otherList.componentType);
            }
            return false;
        }

        public CatscriptType getComponentType() {
            return componentType;
        }

        @Override
        public String toString() {
            return super.toString() + "<" + componentType.toString() + ">";
        }
    }

}
```

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

Catscript Guide

This Document should serve as a guide for new users coding in catscript

Introduction

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

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

output: foo
```

Features

Here we will disect each section of the grammar and provide examples. To start, you need a program.

Catscript Program

```
catscript_program = { program_statement };
```

A catscript program consists of one thing, a program statement. This means in order to have a catscript program compile, you only need a program_statement!

Program Statement

```
program_statement = statement |
                    function_declaration;
```

A program statement consists of a statement or a function_decleration. A valid program statement consits of all statements (if, for, print etc) or a function decleration (see below).

Statement

```
statement = for_statement |
if_statement |
print_statement |
variable_statement |
assignment_statement |
function_call_statement;
```

A catscript statement is simply one of the statements that catscript supports. This includes for statement, if statement, print statement, variable statement, assignment statement, and function call statement. See below for details on individual statements

For Statement

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
                '{', { statement }, '}';
```

A catscript for statement consists of a IDENTIFIER and expression, with a statement in the body. See below for details on IDENTIFIER and expressions. The catscript for statement takes the IDENTIFIER and matches it to the expression, and then in the body executes the expression. In the example below, the IDENTIFIER i would be matched to the expression [1,2,3], and the statement print(i) would in this case print integers 1,2,3

```
//example for loop
for(i in [1,2,3]){
    print(i)
}
```

If Statement

```
if_statement = 'if', '(', expression, ')', '{',
                    { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

The if statement consists of an expression, with a statement in the body, followed by an optional else, with an additional if statement in the else body. If statements are used to execute statement based off of the expression. In the example below, the if statement compares 2 values, if the expression is true, then the statement is executed, if it is false, then the else statement is executed. Additionally, you can stack multiple if statements on top of each other using else if, example below.

```
// if statement example
var x = 10
if(x > 8){
    print("x is greater than 8")
}
// output: x is greater than 10

// if else example
if(x > 11){
    print("x is greater than 11")
}else{
    print("x is less than 11")
}
//output: x is less than 11

// if else if example
if(x < 9){
    print("x is less than 9")
}else if(x == 10){
    print("x equals 10")
}else{
    print("x is greater than 9 but does not equal 10")
}
//output: x quals 10
```

Print Statement

```
print_statement = 'print', '(', expression, ')'
```

The print statement is a simple statement that prints out an expression. You can use print statements in catscript to print out any valid expression.

```
// print statment examples
// print string
print("hello world")
//output: hello world

//print integer
print(1000)
//output: 1000
```

Variable Statement

```
variable_statement = 'var', IDENTIFIER,
     [':', type_expression, ] '=', expression;
```

In catscript you use variable statements to declare variables. Use var IDENTIFIER to declare the variable name, followed by an optional ':' type, then an = expression. You can optionally identify the type, or the type can be inferred, see type_expression below. In catscript everything is assignable to object.

```
//variable statement examples
// implicit type
var x = "foo"

var x = 1000

// explicit type
var x : integer = 10

var x : object = 10
```

Assignment Statement

```
assignment_statement = IDENTIFIER, '=', expression;
```

Assignment statement, similar to variable statement, assigns an identifier to an expression. Simply have an identifier, an =, and a valid expression. If the identifier is not in scope, then an error will occur. Use assignment statements to reassign values to scoped variables

```
// assignment statement example
var x = 9
x = 10

var x = "fee"
```

```
x = "foo"
```

Function Declaration

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                       [ ':' + type_expression ], '{',  { function_body_statement },  '}';
```

Use function declaration to define your functions in cat script. To define a function use an IDENTIFIER, an optional paramater_list, an optional type, and a body of function_body_statements. See function body statements below. Once you have defined a function, you can call it using a function_call statement, see below for function call.

```
// function declaration examples
function foo(a,b){
    a + b
}

function fee() : string {
    print("fee fi fo")
}
```

Function Call Statement

Function calls are used to call a function after it has been defined. Use function calls to execute your function later in your catscript program. For this example, assume you have declared your functions like in the function_declaration example.

```
// function call example
foo(1,2)
//output: 3
fee()
//output: fee fi fo
```

Function Body Statements

```
function_body_statement = statement |
                          return_statement;
```

Function body statements consist of a statement, or a return_statement. See return statement below.

```
// function body statement example
function foo(a,b){
    a + b
}
```

Return Statement

```
return_statement = 'return' [, expression];
```

Return statements are used in function definitions to return the desired value of the function. Return can also be used to break out of a function at its current state. Returns can optionally be followed by any valid expression.

```
// function with return expression
function multiply(a){
    return a * a
}

// function return
function checkEqual(a,b){
    if(a == b){
        return true
    }else{
        return
    }
}
```

Paramaters

```
parameter = IDENTIFIER [ , ':', type_expression ];
parameter_list = [ parameter, {',' parameter } ];
```

A parameter is an IDENTIFIER name with an optional ':' type expression. See type expression below A parameter list is a list of parameters, with optionally multiple parameters seperated by commas

```
// parameter examples
x
x : integer

// parameter list examples
[x, y, z]
[ x : integer, y : integer, z : integer]
```

Equality Expression

Equality expressions are compare two values using == for equals, or != for not equal

```
// equality expression examples
if( x == y){
    statement
}
if(x != y) {
    statement
}
```

Comparison Expression

Comparison expressions compare two values using >, >=, <, <= (greater, greater than or equal, less than, less than or equal)

```
//comparision example
if(x > 0){
    ...
}
if(x >= 0){
    ...
}
if( x < 0) {
    ...
}
if( x <= 0 ) {
    ...
}
```

Additive Expressions

Additive expressions are used for addition and subtraction, additive expressions are left associative and can be parenthesized

```
//additive expression examples
1 + 1
x + 1
1 + (1 + 1)
//subtraction examples
1 - 1
x - 1
1 + (1 - 1)
```

Factor Expressions

Factor expressions consist of multiplication and division. Use * for multiplication and / for division. Catscript has standard order of operations and factor expressions can also be parenthesized.

```
//multiplication examples
1 * 1
1 * 2 * 3
1 * (1 * 2)
//division examples
10 / 5
10 / 5 / 2
10 / (5 * 2)
```

Unary Expressions

Unary expressions are negative numbers or "not". This allows users to negate any value. You can also have nested unary expressions

```
//unary expression examples
x = -10
y = not true
y = not not true
```

Primary Expressions

Primary expressions are the basis of the catscript grammar, all expressions will recursively get to primary expression to express a given value.

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null"|
                     list_literal | function_call | "(", expression, ")"
```

Literal Expressions/ Type Expressions

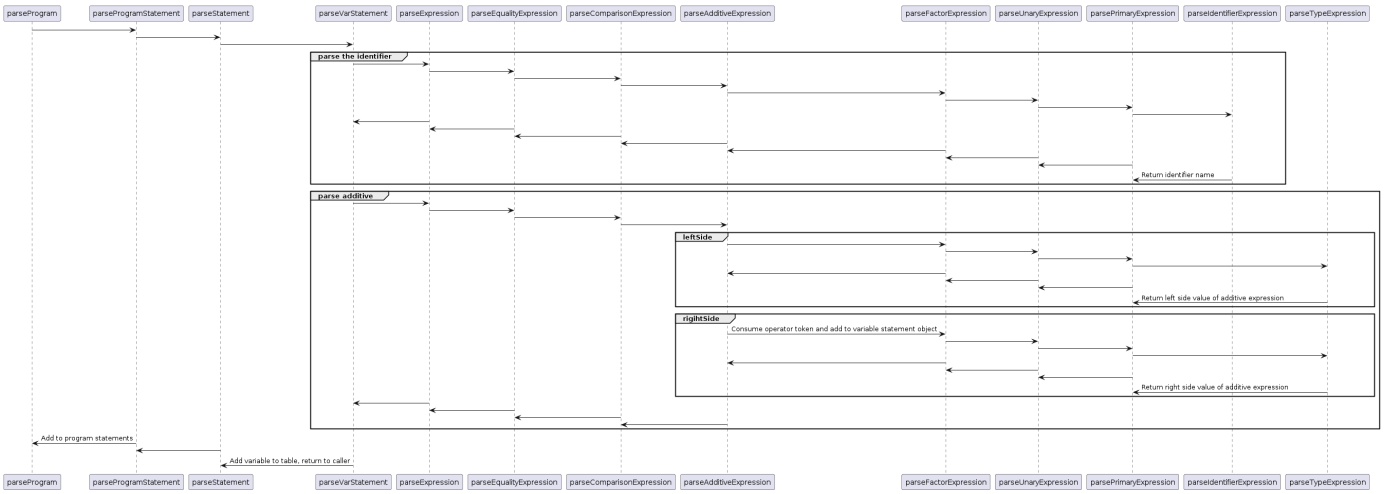
Type (or literal) expressions are the basic types of Catscript, they are the basic primitive types as well as List and Object. Lists can optionally have their own types. Nulltype is assignable to everything, and everything is assignable to an Object type

```
var boolExample = true
var intExample = 10
var stringExample = "stringExample"
var nullExample = null

var implicitList = [1,2,3]
var explicitList : list<object> = [1,2,3]
```

Section 5: UML.

This UML diagram is a sequence diagram illustrating the recursive decent nature of our parser. The statement to be parsed here was a variable statement which I set to an additive expression which would need to recursively parse both the left and right side of the expression. The values that are create are then pushed up to our statement and then to our program. The tokens that are created help are consumed



Section 6: Design trade-offs

There are two main schools of thought for compilers currently in the industry. For the most part compilers are design as a recursive decent parsers which are top down compilers whereas parser generators are the opposite and are bottom up. Bottom up parsers still can be popular though as seen with the YACC which stands for Yet Another Compiler Compiler which is used in unix systems.

First lets start off with parser generators which we did not choose for this project. Parser generators can create really powerful grammar but overall it is harder to implement, but it is to be noted that the testing and development may get harder for the more complicated grammars that occur with a Recursive Decent Style compiler.

Looking at the recursive descent parser the implementation is vastly more simplified and allows users to successfully implement large amounts of grammar with reality ease being a much more useful first project choice. The industry also is by far has more development and use of recursive decent parsers which makes this is more practical project and allows for troubleshooting and debugging to be simplified as more users have experience in this field.

Section 7: Software development life cycle model

Test Driven Development (TDD) was utilized for this project. This sort of development greatly assisted the team. There are many approaches that could have worked in this scenario but the test driven development had a few specific elements that make this the optimal choice. Firstly Test Driven Development allows programmers to outline the project into sensible progress checkpoints. This allows for programmers to easily set goals break up work and create metrics to measure developmental success. In my experience this makes time lining the project significantly easier.

When looking at other considerations like debugging Test Driven Development is king in this arena, this allows users to test specific functions of the program and allows other team members to easily debug as well. Other test based developments also take advantage of this as well, and it should be noted that those are all subcategories of this. A popular subcategory of test driven development is tested first but this is not as effective and therefore isn't recommended. Test first development is used to some degree but that is only because of great work in designing the parser beforehand, in most cases this is not ideal, but it worked quite well in this project.

When looking at other development life cycles there are popular ones that were taught here in software design those being waterfall and agile. For the most part these are great ways to accomplish some of the work being done here in the project but because of the team structure and the development that we were expected to do and time limitations Test Driven Development is a more reasonable way to conquer this.