

# Compilers CSCI 468

Hunter R. Hayward

## Synopsis:

- This portfolio contains my capstone project for my bachelors of science degree at Montana State University. The capstone project was to write a recursive descent parser for a language given some grammar and specifications. For this project the language was Cat script and we used java to write the parser. For the development life cycle of the project test driven development was used for this project. This portfolio contains all standard industry practices that a computer scientist should be familiar with. That includes UML diagrams and writing the documentation for Cat script. This project is an independent project. However there is a section on creating unit tests for our partners to use in their code. Other than that everything was done independently.

## Section I

Here are the Git buh repositories. The file with the code should be included local as well.

<https://github.com/msu/csci-468-spring2022>

<https://github.com/hhayward98/csci-468-spring2022-private>

## Section II

TeamMembers:

Since this project was primarily an independent project, there has not been much team work done. We did have to partner up for the last part of the project. Based on my partner's code I would say they have put in some effort but probably should spend more time with a tutor. I can say I was in the same boat as my partner. My partner and I did collaborate on some of the tests we needed to get passed.

Part of the assignment was that we write unit tests for our partners' code. My partner wrote me these test below.

```
@Test
public void unitT1() {
    AdditiveExpression expr = parseExpression("1 + (1 * 1)");
    assertTrue(expr.isAdd());
}
```

```
@Test
public void unitT2() {
    ComparisonExpression expr = parseExpression("1 < (1+1)");
    assertTrue(expr.isLessThan());
}
```

```
@Test
public void UnitT3() {
    VariableStatement expr = parseStatement("var h : int = 5");
    VariableStatement expr1 = parseStatement("var x : int = 10");
    assertEquals(expr1.getExplicitType(), expr.getExplicitType());
}
```

For the first test unitT1. The purpose was to check if the parser can handle multiplication and addition, while handling parentheses. It is important that the parser can handle basic math.

For unitT2 the purpose was to check if the parser can compare and evaluate properly. Having a parser that can correctly evaluate before comparing is important.

Finally for unitT3 the test is checking the types of variables. I think this test is designed to show the type is consistent even with different values.

## Section III

The Design pattern I decided to use is the memoization I had to implement within the code. Here is what the code looks like :

```
private static Map<CatscriptType, ListType> cache = new HashMap<>();

public static CatscriptType getListType(CatscriptType type) {

    ListType listType = cache.get(type);

    if(listType != null){

        return listType;

    } else{

        ListType nLitstype = new ListType(type);

        return nLitstype;

    }

}
```

Memoization in this example is not thread safe. There are issues with this example but, Since Cat Script is single threaded. There is no need to worry about issues with multi threads and runtime errors.

Memoization makes sure that a method does not run the same inputs more than once. This is done by keeping a record of results based on inputs. By doing this the parser is able to anticipate your code and return the stored outputs rather than process the input again. For example, if given variable x equal to five and variable y equal to four. If we run a multiply function that takes two integer arguments and multiplies them. Memoization will store the output of the function associated with the input values of x and y. So, if x is multiplied by y. The output should be twenty. Thus, when input values for x are five and y is four. Instead of multiplying the two values, the memoization will anticipate the output using the stored data and just return the stored value for output.

## Section IV

# Documentation

Here is the Documentation for cat script. Below I have sectioned off the different parts of my parser based on what is being done.

## Grammar:

catscript\_program = { program\_statement };

program\_statement = statement |  
                    function\_declaration;

statement = for\_statement |  
            if\_statement |  
            print\_statement |  
            variable\_statement |  
            assignment\_statement |  
            function\_call\_statement;

for\_statement = 'for', '(', IDENTIFIER, 'in', expression ')',  
                '{', { statement }, '}';

if\_statement = 'if', '(', expression, ')', '{',  
                { statement },  
                '}' [ 'else', ( if\_statement | '{', { statement }, '}' ) ];

print\_statement = 'print', '(', expression, ')'

variable\_statement = 'var', IDENTIFIER,  
                    [':', type\_expression, ] '=', expression;

function\_call\_statement = function\_call;

assignment\_statement = IDENTIFIER, '=', expression;

function\_declaration = 'function', IDENTIFIER, '(', parameter\_list, ')' +  
                    [ ':' + type\_expression ], '{', { function\_body\_statement }, '}';

function\_body\_statement = statement |

```

return_statement;

parameter_list = [ parameter, {',' parameter } ];

parameter = IDENTIFIER [ ':', type_expression ];

return_statement = 'return' [, expression];

expression = equality_expression;

equality_expression = comparison_expression { ("!=" | "==") comparison_expression };

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" )
additive_expression };

additive_expression = factor_expression { ("+" | "-" ) factor_expression };

factor_expression = unary_expression { ("/" | "*" ) unary_expression };

unary_expression = ( "not" | "-" ) unary_expression | primary_expression;

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

list_literal = '[' , expression , { ',' , expression } ']';

function_call = IDENTIFIER , '(' , argument_list , ')'

argument_list = [ expression , { ',' , expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression , '>']

```

## Statements:

**Print Statement:** Prints out the value inside of the parentheses

Print Statement: `print(1)`

**Assignment Statement:** assigns a value.

Null case Example: `x = null;`

Example: `x = "foo"`

**Return Statement:** Returns a value/expression.

Return in Function Example:

- Without Expression: `function x() {return}`
- With Expression: `function x() : int {return 10}`

Basic Example: `return x`

**For Statement:** iterates through its body statement for a given/set amount of iterations. With some Identifier representing the current iteration within the given range or set of values.

Example: `for(x in [1, 2, 3]){ print(x) }`

**If Statement / Else Statement:** Conditional statement. Conditions are set to trigger code for situations where code is needed. The If statement can be called by itself and can take 1 value as a parameter but will evaluate as a boolean. The else clause requires there to be an existing if statement that is not closed with a else statement already.

IF Example: `if(x > 10){ print(x) }`

IF / Else Example: `if(x > 10){ print(x) } else { print( 10 ) }"`

**Boolean Statement:** The boolean statement is just a variable statement set to a bool value.

Example: `var x : bool = true`

**Variable Statement:** the variable statement is just like the assignment statement. Except here we are declaring a variable by using “var” then passing the variable name followed by a colon and the variable type, all set equal to some value given by the user. Unless the variable is an explicit type. Which means no type was declared, and the variable will be assigned the most logical type based on value passed in.

Var with explicit type: `var x = 10`

Var with int type: `var x : int = 10`

Var with string type: `var x : string = “hi”`

Var with object: `var x : object = foo()`

Var with list type: `var x : list<int> = [1, 2, 3]`

## Expressions:

The following expression examples show how the parser handles different expressions.

### Additive Expressions:

Add: `1 + 1`

Subtract: `1 - 1`

### **Multiply/divide Expressions:**

Multiply: `1 * 1`

Divide: `1 / 1`

**Comparative Expressions:** In Cat script when we compare two objects it's just like most other programming languages. There is a right hand side value and a left hand side value. These two values are evaluated and compared to test for conditions set by the user.

Greater: `1 > 1`

Less Than: `1 < 1`

Greater Than or Equal to: `1 >= 1`

Less Than or Equal to: `1 <= 1`

**Equality Expression:** The equality expression is the exact same as the comparative expression for evaluating both right and left hand sides of the operator. The main difference is the equality expression test only if both values are equal.

Example: `1 == 1`

**Unary Expressions:** Same as equality expression but NOT equal instead.

Example: `1 != 2`

**Boolean Expressions:** Either true or false.

Example: `true`



## List Literal Expression:

Example: `[1, 2, 3]`

## Functions:

**Statements:** the function call and function definition statements are used to create and call functions in catscript. For defining a function it starts with typing “function” followed by the name of the function and parentheses wrapping the parameters being read into the function. Then within curly braces is the function body. The body contains statements the function needs to execute. So, when we call the function the function body is evaluated based on any parameter passed in if any are.

- Function call Statement: `x(1, 2, 3) y = 1`
- Function Definition Statement:
  - No parameters: `function x() {}`
  - With parameters: `function x(a, b, c) {}`
  - With parameter types: `function x(a : object, b : int, c : bool) {}`

**Expressions:** The function call expression just shows how the cat script parser reads functions.

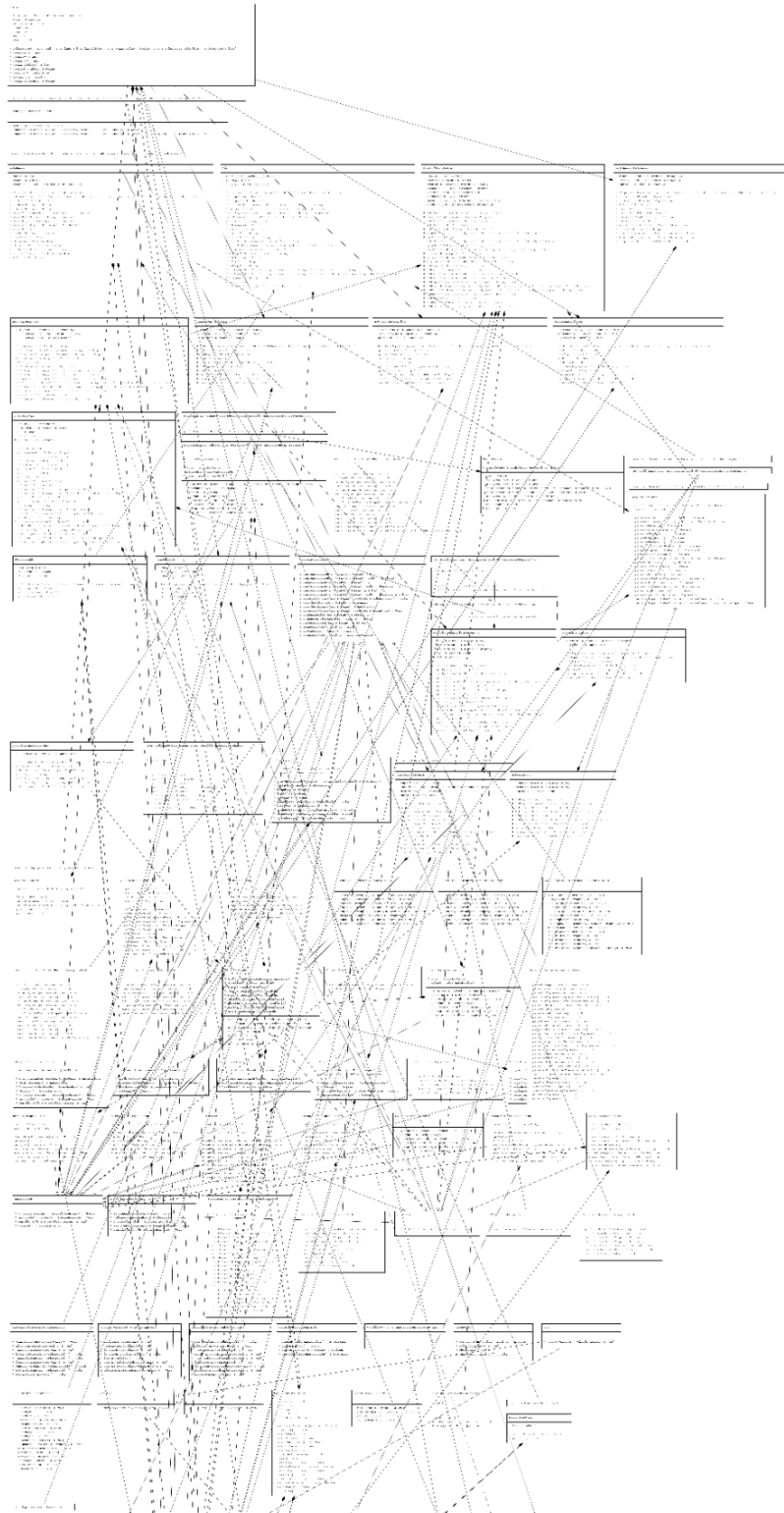
- Function call Expression: `foo(1, 2, 3)`
  - In the example above the function that is being called looks like the following: `function foo(a, b, c) {}`
  - The parameters a, b, c, are 1,2,3 in the function call example just above.

## Section V

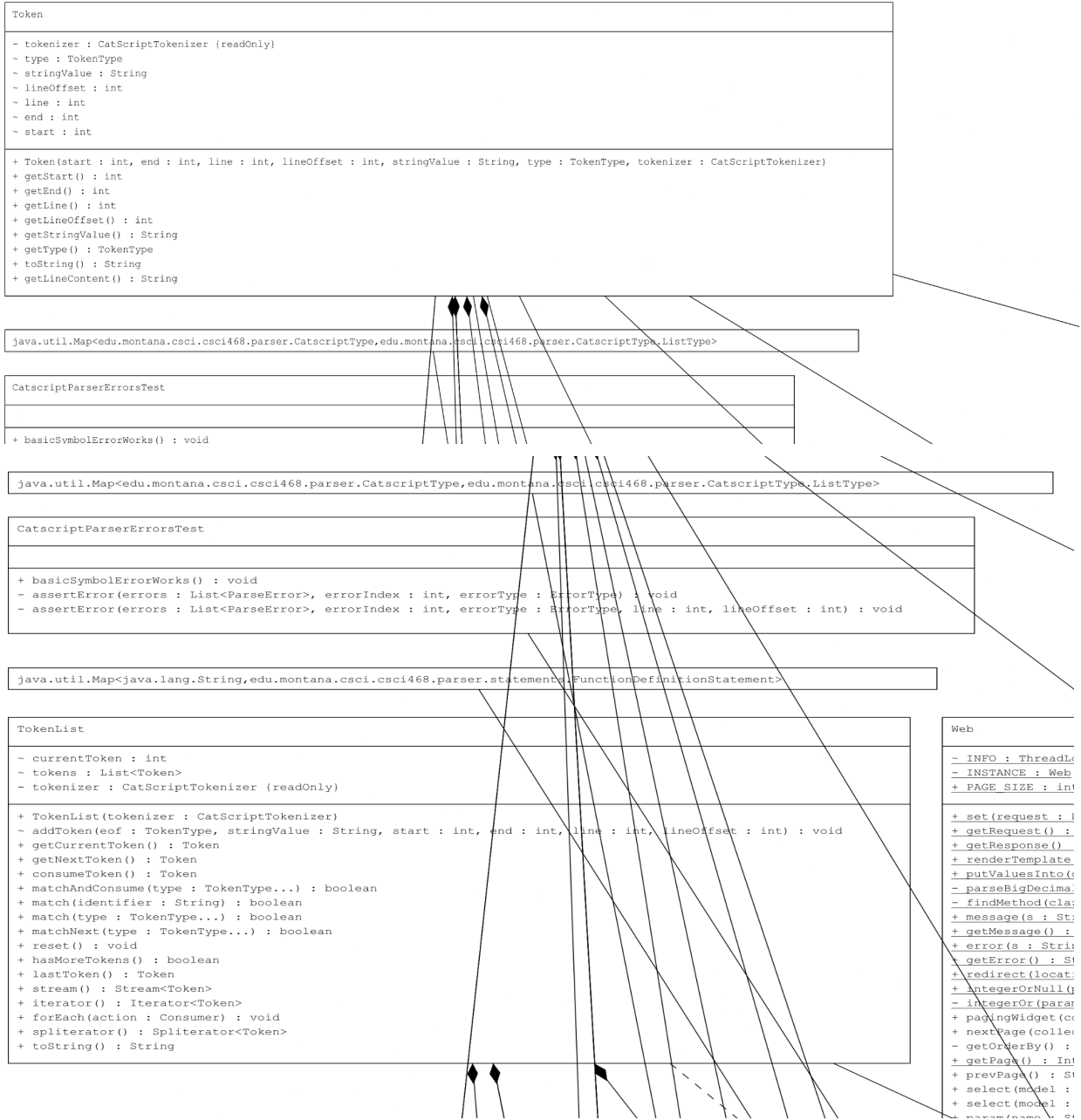
**Below is the main UML for the Code**

I had to use a lot of objects and most free websites dont allow that many objects. I have the main image below but I have also included blown up images of the different parts of the UML diagram that are important and relevant to what I actually coded.

The image below is Titled image 1.0



@here Starts at Top Left of image 1.0



ParseElement
- errors : List<ParseError> - children : List<ParseElement> - end : Token - start : Token # parent : ParseElement
+ ParseElement() + getProgram() : CatScriptProgram + setStart(start : Token) : void + setEnd(end : Token) : void + setToken(token : Token) : void + getParent() : ParseElement + getStart() : Token + getEnd() : Token + getErrors() : List<ParseError> + hasErrors() : boolean + addError(errorType : ErrorType, args : Object...) : void + addError(errorMessage : ErrorType, token : Token, args : Object...) : void # addChild<T> (element : T) : T + getChildren() : List<ParseElement> + toString() : String + hasError(errorMessage : ErrorType) : boolean - registerFunctions(symbolTable : SymbolTable) : void + verify() : void {readOnly} + validate(symbolTable : SymbolTable) : void - collectErrors(collector : LinkedList<ParseError>, parseElement : ParseElement) : void + transpile(javascript : StringBuilder) : void + compile(code : ByteCodeGenerator) : void # box(code : ByteCodeGenerator, type : CatscriptType) : void # unbox(code : ByteCodeGenerator, type : CatscriptType) : void

ParseErrorInfo

java.util.LinkedList<edu.montana.csci.csci468.parser.ParseError>

java.util.LinkedList<java.util.Map<java.lang.String, java.lang.Object>>

java.util.List<edu.montana.csci.csci468.parser.Expression>

FunctionCallExpression

- type : CatscriptType  
 - arguments : List<Expression>  
 - name : String {readOnly}

+ FunctionCallExpression(functionName : String, arguments : List<Expression>) : void  
 + getName() : String  
 + getType() : CatscriptType  
 + validate(symbolTable : SymbolTable) : void  
 + evaluate(runtime : CatscriptRuntime) : Object  
 + transpile(javascript : StringBuilder) : void  
 + compile(code : ByteCodeGenerator) : void

FunctionCallStatement
- expression : FunctionCallExpression
+ FunctionCallStatement(parseExpression : FunctionCallExpression) + getArguments() : List<Expression> + validate(symbolTable : SymbolTable) : void + getName() : String + execute(runtime : CatscriptRuntime) : void + transpile(javascript : StringBuilder) : void + compile(code : ByteCodeGenerator) : void

java.util.List<edu.montana.csci.csci468.parser.ParseError>

ClosuresDemo

+ closureDemo1() : void  
 + createAdder() : Callable<Integer>  
 + closureDemo2() : void  
 + createAdderWithLambda() : Callable<Integer>  
 + createAdderWithLambda2() : Callable<Integer>  
 + listMappingWithLambda() : void  
 + listMappingWithObjects() : void  
 + filterTest() : void  
 + filter<T> (values : List<T>, filter : Predicate<T>) : List<T>

UnaryExpression

- rightValue

+ operator

+ UnaryExpression(rightValue : Object, operator : String)

+ getRightValue() : Object

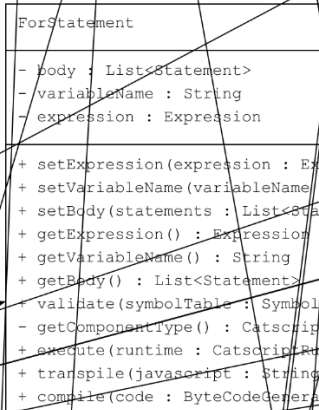
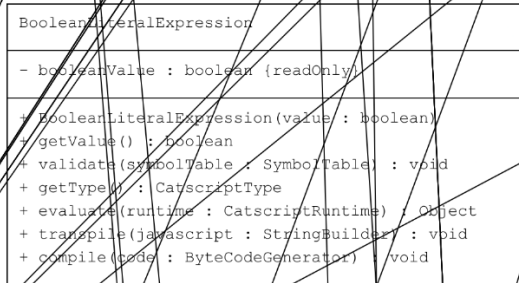
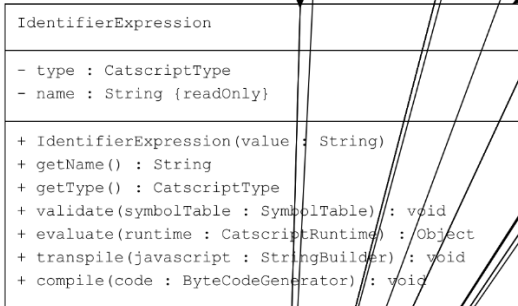
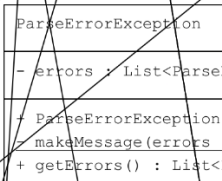
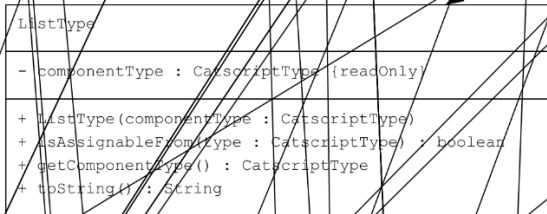
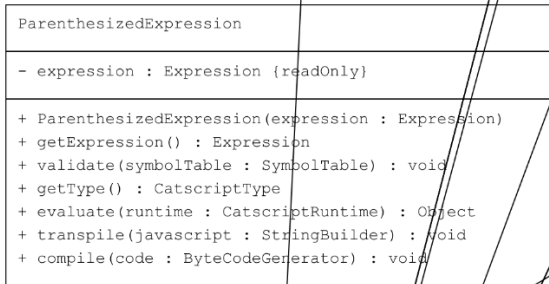
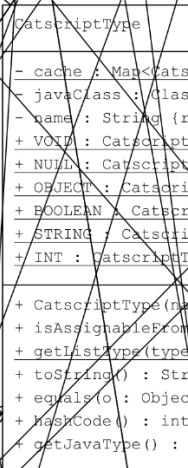
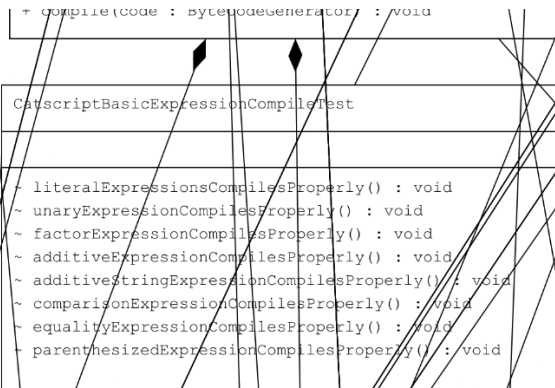
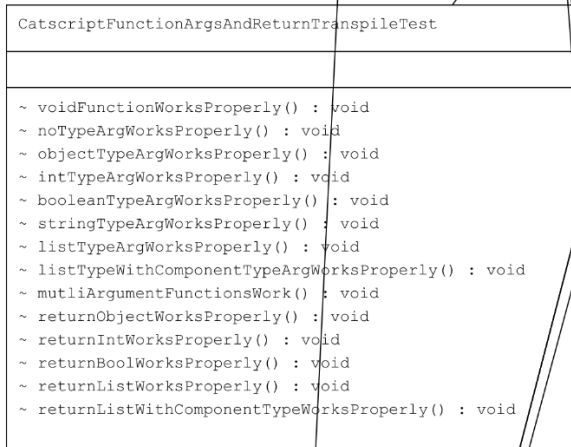
+ getOperator() : String

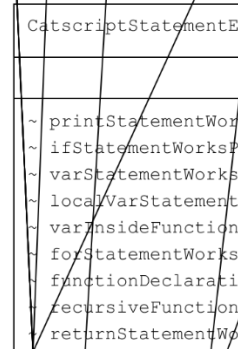
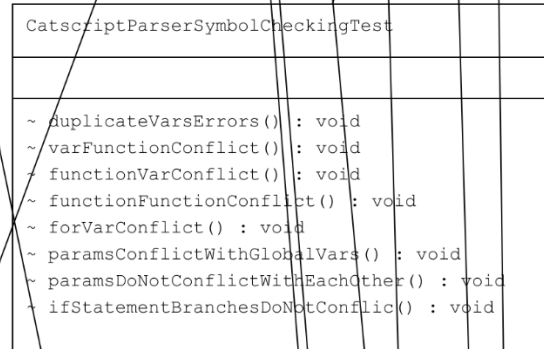
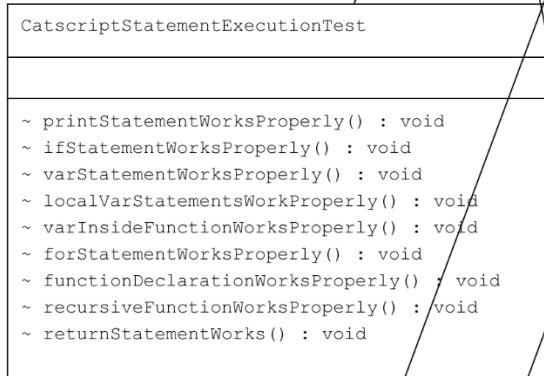
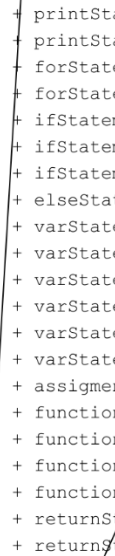
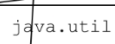
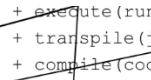
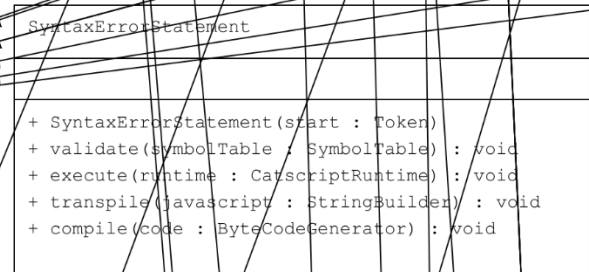
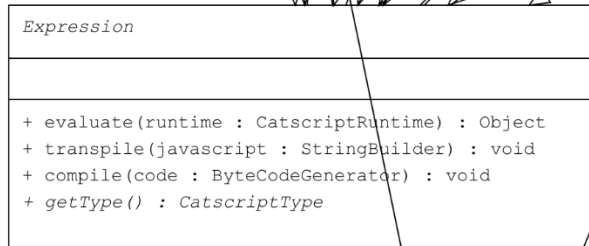
+ validate(symbolTable : SymbolTable) : void

+ evaluate(runtime : CatscriptRuntime) : Object

+ transpile(javascript : StringBuilder) : void

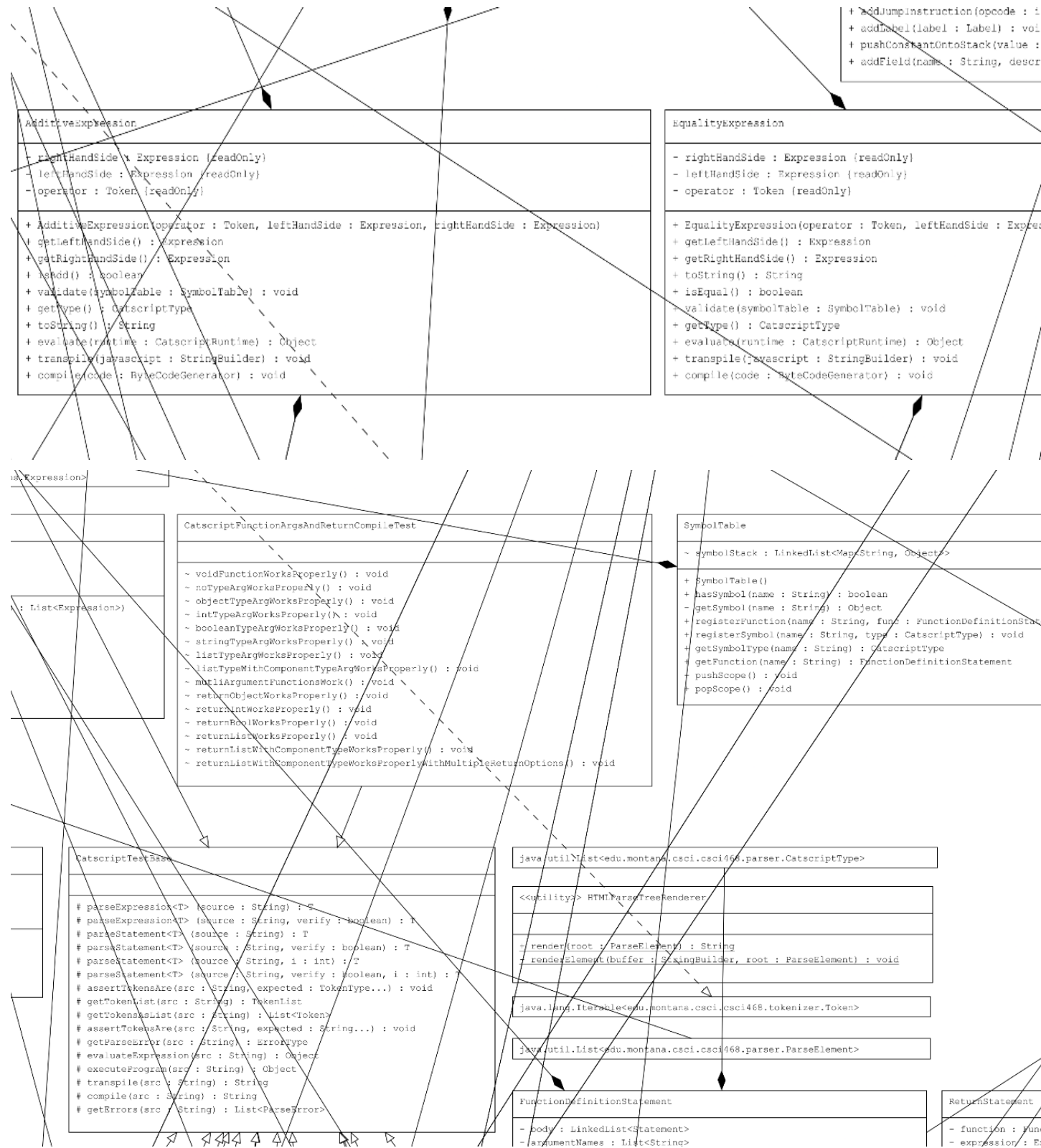
+ compile(code : ByteCodeGenerator) : void







The following is the important stuff from the middle





ParseError>

>

cat<T> : List<T>

UnaryExpression

- rightHandSide : Expression (readOnly)  
- operator : Token (readOnly)

+ UnaryExpression(operator : Token, rightHandSide : Expression)  
+ getRightHandSide() : Expression  
+ isMinus() : boolean  
+ isNot() : boolean  
+ toString() : String  
+ validate(symbolTable : SymbolTable) : void  
+ getType() : CatscriptType  
+ evaluate(runtime : CatscriptRuntime) : Object  
+ transpile(javascript : StringBuilder) : void  
+ compile(code : ByteCodeGenerator) : void

java.util.List<edu.montana.csci.csci4

CatscriptProgram

- expression : Expression  
- functions : Map<String, FunctionDef  
- statements : List<Statement>  
- output : StringBuffer

+ print(v : Object) : void  
+ getOutput() : String  
+ addStatement(child : Statement) : void  
+ setExpression(expression : Expression)  
+ getExpression() : Expression  
+ getStatements() : List<Statement>  
+ isExpression() : boolean  
+ getFunction(name : String) : FunctionDef  
+ validate(symbolTable : SymbolTable)  
+ execute() : void  
+ execute(runtime : CatscriptRuntime)  
+ transpile(javascript : StringBuilder)  
+ compile(code : ByteCodeGenerator) :

PrintStatement

- expression : Expression

+ setExpression(parseExpression : Expression) : void  
+ getExpression() : Expression  
+ validate(symbolTable : SymbolTable) : void  
+ execute(runtime : CatscriptRuntime) : void  
+ transpile(javascript : StringBuilder) : void  
+ compile(code : ByteCodeGenerator) : void

CatscriptBasicExpressionEvalTest

~ literalExpressionsEvaluatesProperly() : void  
~ unaryExpressionsEvaluatesProperly() : void  
~ factorExpressionsEvaluatesProperly() : void  
~ additiveExpressionsEvaluatesProperly() : void  
~ additiveStringExpressionsEvaluatesProperly() : void  
~ comparisonExpressionsEvaluatesProperly() : void  
~ equalityExpressionsEvaluatesProperly() : void  
~ parenthesizedExpressionsEvaluatesProperly() : void

CatscriptBas

~ literalExp  
~ unaryExpre  
~ factorExpr  
~ additiveEx  
~ additiveSt  
~ comparison  
~ equalityEx  
~ parenthesi

## Section VI

### Parse Generators:

- Parse generators are programs that generate a parser based on specifications given by the user. There are many types of parser generators used in industry everyday. Parse generators cut the time of creating a parser down significantly. Using parse generators can be very useful for certain tasks. I think parse generators are cool but should be used by people who fully understand how to write a parser by hand.

### Recursive descent:

- Recursive descent parser is a top-down parser that consists of mutually recursive procedures. Each procedure implements a nonterminal from the grammar. This makes the structure of the program mirror the grammar that it was designed after. Top-down recursion means the parser starts at the top and flows down. This means the order of what is being parsed by the parser is very important. As some expressions and statements have dependent variables that change depending on logic and input values. I Believe doing recursive descent parsing was way more valuable than learning how to use a program that does everything, that's like taking a class on using an IDE. That is why I think writing a parser by hand is the best thing for students to do. Since we are students we should be learning about how things work on a deep level. There are situations where using a parser generator might be the most practical and optimal route. In most situations writing a parser by hand is the best way to go especially if it has complex language specifications.

## Section VII

For the capstone project we used Test Driven Development (TDD) as our model. This means prior to creating the actual program. Tests are created to check if the code is working correctly. At first this was hard to comprehend. It reminded me of the chicken or the egg problem.

The Tests that are created and used in the TDD have to be generic to work properly. For our environment the tests were formatted following this pattern.

```
public void additiveExpressionsCanBeParenthesized() {  
  
    AdditiveExpression expr = parseExpression("1 + (1 + 1)");  
  
    assertTrue(expr.isAdd());  
  
    assertTrue(expr.getLeftHandSide() instanceof IntegerLiteralExpression);  
  
    assertTrue(expr.getRightHandSide() instanceof ParenthesizedExpression);  
  
}
```

The “assertTrue” method tests to see if the expression is true. If it is true then the test will pass, else the test will fail. This process for testing code is very useful. We can write test for code that hasn't been written based on what our output should be.

Using TDD has been very valuable for me. I know how to use this type of development to create my own projects. By using tests and assertions I can debug my future code without having to write it. As long as I know what I need to my code to produce or return, ill be able to test my output until I get my desired results. This skill is invaluable to my development and I hope to see this type of technique used in my future if any needs to be used at all.