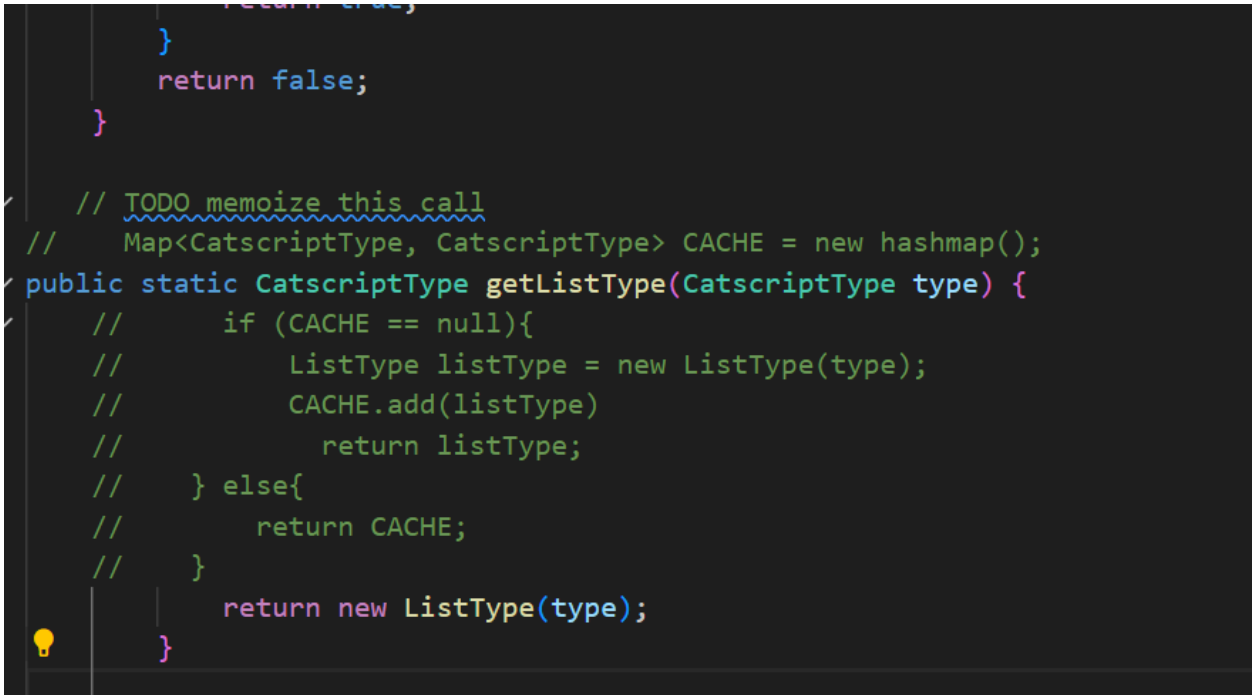
Compilers
CSCI 468
Spring 2024
Aden Hartman & Willow Berryessa

Program:

Teamwork:

Aden Hartman was the tester for my capstone. His primary contributions included 3 tests to help debug and refine my code as well as the documentation provided in section 4. Overall, his primary contribution to this project took around 5% of the total time taken to complete this project. I, Willow Berryessa, was responsible for the implementation of the catscript tokenizer, parse, and compiling methods. My contribution to this project handled the majority of the time taken to complete this project totaling around 95%.

Design Pattern:
The attachment below is my implementation of memoization which is used to optimize the performance of functions which I implemented in the commented out lines. I chose to use this as my example of a design pattern because it was the recommended one and because it is a great example of something to implement in the future.



Technical Writing:
Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

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

```
```

```
var x = "foo"
print(x)
```
Below in the Features sections are all the different Statements and Expressions possible in the catscript language and some examples of them.

Features - Expressions and Statements

statement =
        for_statement   //Statement&Expression Composition as described in class repo
        if_statement   //Example usages and descriptions provided
        print_statement
        variable_statement
        assignment_statement
        function_call_statement;
Usage Example:
```
for(_ in _){}
if(){}
print();
int x = 0;
x = 9;
woohoo();

```
Description: Each of these statements belong to the statement class, representing different actions to perform on expressions. Some of these statements can be embedded in each other like variable_statement or print_statement.

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')', '{', { statement }, '}';
Usage Example:
```
for(z in [4,2,1]) { print(z) }

```
Description: The for statement is used to iterate. The beginning of the statement is where one declares the name for each iteration(z here). The end of the statement is what will be iterated through, often a list. After declaring what will be iterated through and how, one can declare what they want to happen in the body.

if_statement = 'if', '(', expression, ')', '{', { statement },'}' ['else', (if_statement | '{', { statement },'}')];
Usage Example: "if(true){ print(42) }"
```
```

if(true){ print(42) }

```
Description: The if statement is used to check for a condition before executing the body. The condition is found in the parentheses and can be a boolean or something that evaluates to a boolean(comparison).

## print_statement = 'print', '(', expression, ')'
Usage Example:
```
print("Sup man");

```
Description: The print statement is used to display information of any type. Parentheses are required for strings, and many objects require toString methods to ensure a proper display.

## variable_statement = 'var', IDENTIFIER, [':', type_expression, ] '=', expression;
Usage Example:
```
var x = 42

```
Description: Variable statements can be used at any scope to store and manipulate values. Different variable types can't be handled together, and must be converted to do so.

## assignment_statement = IDENTIFIER, '=', expression;
Usage Example:
```
boo = "hoo";

```
Description: Assignment statements are used to assign or reassign values to previously initialized variables. These assignments can only be of the same type, or null.

## function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' + [ ':' + type_expression ], '{',  { function_body_statement },  '}';
Usage Example:
```
function woohoo(x, y, z) { print(z) }

```
Description: Function definition statements are used to define a function's name, its parameters, and its body contents. All together functions can be used to handle many values in different ways.
```

function_body_statement = statement | return_statement;
Usage Example:
```
function x() : int {return 10}

```
Description: Function body statements are the soul of the function as they do all the work. Aside from other statements, function bodies can be used to return values as a result of running the function.

return_statement = 'return' [, expression];
Usage Example:
```
function x() : int {return 10}

```
Description: Return statements are used to return a value as the result of a function. Return values can be any type but must align with the other types.

equality_expression = comparison_expression { ("!=" | "==") comparison_expression };
Usage Example:
```
1 == 1

```
Description: Equality expressions are used to compare values. Whether it be == or !=, these expressions can only compare similar types and require a compareTo for objects.

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=") additive_expression };
Usage Example:
```
x > 4

```
Description: Comparison expressions are also used to compare values, but have more range in their capabilities. Additionally, these expressions can only be used on integers or variables representing such.

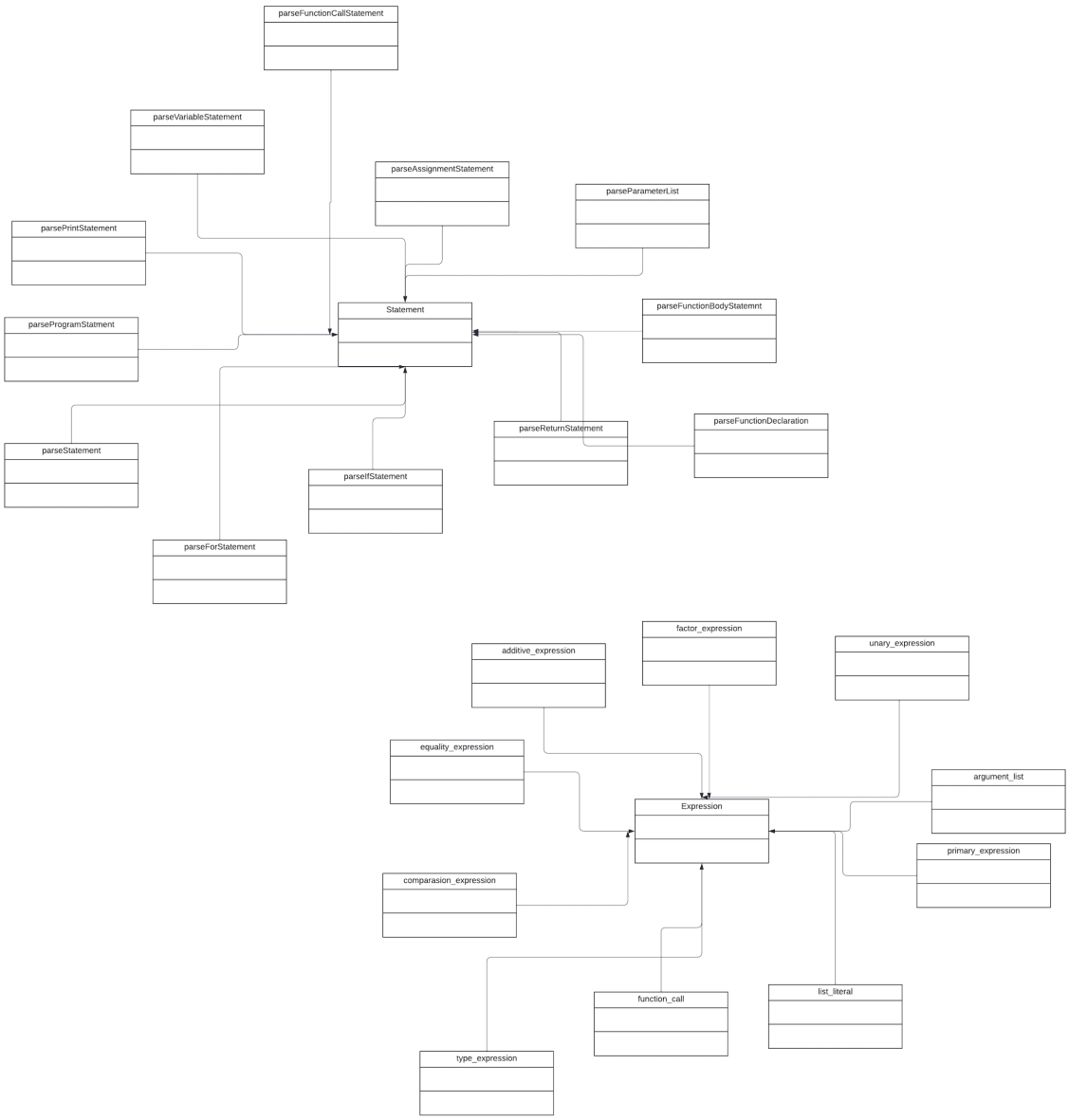
additive_expression = factor_expression { ("+" | "-") factor_expression };
Usage Example:
```
9 + 10
```

```
Description: Additive expressions are used to change values via signs. These expressions can only be performed on ints or strings. Thes expressions can be embedded even in comparison expressions

### factor_expression = unary_expression { ("/" | "*" ) unary_expression };
Usage Example:
```
4 * 20

```
Description: Factor expressions are similar to additive expressions in that they are used to change values via signs. These expressions can only be performed on integers and can also be embedded deeply.

### unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
Usage Example:
```
!true

```
Description: Unary expressions are used to make a value opposite of what it is. They can only be used on expressions that evaluate to be boolean.

### function_call = IDENTIFIER, '(', argument_list , ')'
Usage Example:
```
woohoo(a,b,c);

```
Description: Function call expressions are used to retrieve a value from a function. These expressions evaluate to be whatever type the function dictates, and can be used as values.


UML Section:
```



Design Trade-Offs:

When deciding whether to use recursive descent or a parser generator much was taken into consideration. Recursive descent offered things like a more intimate understanding of the parser because it was being written by myself. A parser generator would not be able to offer me this. However, a parser generator would take up much less time to build and automates most of the parsing process. The recursive descent method also offered very efficient and organized run time. The last thing we had to think about was how strict we wanted our grammar implementation to be. The recursive descent method allowed us to be precise in our implementation whereas a parser generator may have allowed grammar rules to slip by for the

luxury of convenience. After thinking things over we went with the recursive descent option. This option was not only really cool to implement but overall the better option for this project.

Software Life-Cycle:

For this project we used Test Driven Development or TDD. This development process means we were given tests to run to ensure our methods for implementing the catscript grammar were correct. I personally thought this model helped my team to complete this project. Having different tests ensured we were correctly implementing the grammar for all possible cases. The tests also allowed a visual representation of how we were doing as well as something to debug when we got stuck. Overall, I quite enjoyed how this class was set up and appreciated the use of TDD.