Section 1: Program

Link to the zip file containing my capstone project.

Section 2: Teamwork

With my partner's Catscript documentation, there were a lot of important pieces of the documentation already in place. However, there were some big picture / general structure pieces that were missing or needed to be expanded upon. These were the biggest changes I suggested for my partner. In addition to those ideas, I also thought the wording of some sections, especially those about functions needed more details to describe all situations. The main overlooked areas were with explicit typesetting, which my partner fixed after we talked about it. Lastly, I thought that some examples could use comments within the code to explain what was going on to just give additional context with what was going on. Once all of these changes were implemented, I think my partner's documentation ended up being very well fleshed out and fully realized. Their naturalistic tone throughout could have been maybe a little more formal, though that's not a big deal. I still think some explanations could be a little more clear with functions, but they are much better and still work well enough in the final result.

As for my partner's tests, they focused on ensuring type setting is generalized, list types are assigned recursively, and function return coverage is assigned recursively. These tests were well-designed already, but I did suggest a couple changes to improve them. With both of the recursive cases, I recommended adding an extra recursion layer to both tests. To pass the generalization, I checked the type of each object the list stores: if all the items were the same type, that type would be used, otherwise the object type would be used. Next, I had to change my list type setting into a recursive method. I was already using a method called getCatscriptType to find the type of an object, but it wasn't recursive. I changed the method to be recursive when a list type's explicit type is given. Last, I edited my function coverage method to recursively find if each statement has valid return coverage, allowing multiple nested if statements.

In terms of the tests I wrote for my partner, I focused on overlooked functionality and edge cases that weren't covered in the original documentation. For overlooked functionality: I created a test for string addition since not all types of string addition were tested for in the original parsing testing. When strings are added to another type of value, the other value is converted into a string and the two pieces are concatenated together. The original tests covered the addition of strings with ints and null, but lists and booleans are not. I added these two other string addition cases as one of my tests. Next, I made a test to check for edge cases with comma use in lists, function definitions, and function calls. In all three of these, the original test suite does not ensure that there must be commas between every item and there cannot be a comma after the last item. I created a test that made sure this was the case in all three situations. Last, I tested for the edge case of a function with a void return being stored as a variable. Void should not be able to be stored, so my test checks for a type mismatch error when this situation occurs.

Section 3: Design pattern

One design pattern I used in my capstone project can be found in CatscriptType.java in the /src/main/java/edu/montana/csci/csci468/parser directory. The code for this pattern can also be seen copied below:

```
 // Memoization Pattern: when something is expensive to compute based on some arguments
//    So use arguments used to compute expensive thing, and then keep a record
static Map<CatscriptType, CatscriptType> MEMOIZATION_CACHE = new HashMap<CatscriptType, CatscriptType>
public static CatscriptType getListType(CatscriptType type) {

    CatscriptType existingType = MEMOIZATION_CACHE.get(type);
    if (existingType != null) {
        return existingType;
    }
    else {
        MEMOIZATION_CACHE.put(type, new ListType(type));
        return MEMOIZATION_CACHE.get(type);
    }
}
```

I used the Memoization pattern in this specific case for performance reasons. It works by storing the results of an expensive function call in a cache so it can be accessed if the same call is repeated. This doesn't improve performance for the initial call, but for all calls that perform the same action, the performance will be much faster since they can simply access the cache. In this specific case, memoization was used for the getListType method since this method could become expensive if used repeatedly.

To implement this pattern, I started by creating a static map that could be used as the cache for this method. The cache is static for two reasons: the method accessing the cache is also static, and so the cache will be accessible for all CatscriptType objects. Inside the method itself, I altered the method to first check if a value was cached for the type being passed in. If there was already an item in the cache, that can just be used instead of running the expensive method to get a list type. If there was no item in the cache, a new list type is created for it and put into the cache before being returned.

Section 4: Technical writing

Introduction

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

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

General Syntax

Semicolons are not used to denote the end of the line. Instead, instructions need to be separated with white space. This can be a new line as well.

```
 print(x) // new line separates this instruction from the next
print(y)

pring(x) print(y) // white space separated these instructions from one another
```

Comments are started with the slash slash "//", and ended with a new line. * Comments are parts of the code that is not compiled * `// this is a comment that won't be interacted with` * Note that there are no multiline comments in Catscript.

Variables

A variable is a container that holds data. Each variable has a specific data-type. Variables must first be initialized. Then the variable can be set to a new value or used to access the data it holds. Note variables cannot be declared. Instead, they must always hold a value.

Types

The data-type of a variable controls what can be stored, and how it behaves within expressions. * Integers * An integer is a whole number (no decimals). * Strings * A string holds a sequence of characters. * Booleans * A Boolean holds either true or false. * Objects * Can represent any variable data-type, since it is a wrapper type. * Lists * A List is a group of another data type. * A List of objects can store multiple different types.

Initialization

Before a variable can be interacted with, it must be initialized. In which it's data-type is set, and it is given data to hold. Variables can be set implicitly or explicitly. In the following examples, the first is implicit and the second is explicit.

To initialize a variable, the keyword "var" is followed by the variable name. Then by an equal sign, and the value it is being set to. * Variable names must start with either a letter or an underscore "_". * Variable names must be unique. Two variables cannot exist at the same time (see scoping) with the same name.

```
var x = 1
```

Implicit setting

Variables can be initialized implicitly, in which no data-type is defined. Instead, the data-type is assumed from what is being stored.

Explicit setting

Variables can have explicit setting as well. This is when after the name of the variable, there is a collen ":" followed by the keyword for that data type.

TODO: add commements to code (explaining what each example is about)

- Integer
  - The key word is "int"
  - `var x = 6 // implicit var x : int = 6 // explicit`
- Strings
  - The key word is "string"
  - `var x = "abc" // implicit var x : string = "abc" // explicit`
- Booleans
  - The key word is "bool"
  - `var x = false // implicit var x : bool = false // explicit`
- Objects
  - The key word is "object"
  - `var x = 6 // implicit var x : object = 6 // explicit`
- Lists
  - The key word is "list"
  - Lists also have implicit vs explicit typing for the data-type it stores.
  - To set the explicit type, add angle brackets "<>" after the word list, that contain the data type.
  - Lists can also store lists.
  - `var x = [6, 4, 5] // implicit var x : list = [6, 4, 5] // explicit typing of an implicitly typed list var x : list <int> = [6, 4, 5] // explicit typing of an explicitly typed list var x : list <list> = [[6, 4], [2, 8], [1, 5]] // list of lists`

Assigning Variables

Once a variable is initialized, it can be assigned. Use the variable name, an equal sign "=", then the new value it will store.

```
var x = 1   // x is initialized
x = 2       // x is assigned to a new value (in this case 2)
```

Variable Scope

A variables scope determines where a variable can be access, and how long it exists. * Global Variables * These variables are initialized outside of all loops, if statements, and function definitions. * Their lifespan is as long as the rest of the program. * Anywhere after initialization, the variable can be assigned and accessed in all of the program. * Local Variables * These variables are initialized inside of the body of either a loop, if statement, or function definition. * Their lifespan ends when the body it is initialized within ends. * After the variable's lifespan ends, the variable name is free to be used again.

Operations

Operations are a way to interact with other data, such as changing or comparing values.

Arithmetic Operations

Arithmetic operations are done on integer data-types, and return an integer.

When more than one arithmetic operations is in one expression, the operations are completed in PEMDAS order. First, operations with Parenthesis are done. Exponents are not supported in Catscript, so skip that one. Then Multiplication and Division are done. Finally, Addition and Subtraction. If there is more than one type of the same operation, those like items are evaluated in order from left to right. * Addition * Uses the "+" operator * `int x = 12 + 3 // x is set to 15` * Subtraction * Uses the "-" operator * `int x = 12 - 3 // x is set to 9` * Multiplication * Uses the "*" operator * `int x = 12 * 3 // x is set to 36` * Division * Uses the "/" operator * If the resulting number is not a whole number, the decimal is dropped. * ``` int x = 12 / 3 // x is set to 4

```
int x = 10 / 3
// x is set to 3
```

String Operations

- Concatination
    - Uses the "+" operator
    - Creates a new string that contains the string on the left hand side followed by the string on the right hand side
    - Note that string concatenation cannot be done between a string and any other data type. If the other type is not a string, then it is converted into a string.
    - ``` String x = "12" + "3" // x is set to 123

    String x = "12" + 3 // x is set to 123

    String x = "12" + null // x is set to 12null ```

Comparisons

Comparisons compare the values of two different items. Comparison of greater than or less than can only be done between integers. Equality comparisons can be done with any type.

Order of evaluations: * not (on booleans) * less than or greater than * equality * not (if before parenthesis)

Integer Comparisons

- less than
    - Uses the "<" operator
    - Evaluated to true if the left hand side number is less than the right hand side number.
    - `4 < 8 // true 4 < 4 // false 4 < 2 // false`
- less than
    - Uses the "<=" operator
    - Evaluated to true if the left hand side number is less than or equal to the right hand side number. Otherwise, it evaluates to false.
    - `4 <= 8 // true 4 <= 4 // true 4 <= 2 // false`
- less than or equal to
    - Uses the "<=" operator
    - Evaluated to true if the left hand side number is less than or equal to the right hand side number. Otherwise, it evaluates to false.
    - `4 >= 8 // false 4 >= 4 // true 4 >= 2 // true`
- greater than
    - Uses the ">" operator
    - Evaluated to true if the left hand side number is greater than the right hand side number. Otherwise, it evaluates to false.
    - `4 > 8 // false 4 > 4 // false 4 > 2 // true`

Generalized Comparisons

The generalized comparisons of Catscript (equality) can be done between values of any type.

- Equality
    - Uses the "==" operator
    - Evaluated to true if the left hand side value is equal to the right hand side value. Otherwise, it evaluates to false.
    - `4 == 4 // true 4 == 8 // false "hi" == 4 // false null == null // true`

- Not Equality
    - Uses the "!=" operator
    - Evaluated to true if the left hand side value is not equal to the right hand side value. Otherwise, it evaluates to false.

    - `4 != 4 // false 4 != 8 // true "hi" != 4 // true null != null // false`

- Unary
    - Not
    - Uses the keyword "not" before a boolean value or expression to reverse it. True becomes false, false becomes true.
    - If followed by anything other than true or false, it must have parentheses "()" surrounding the following expression.
    - `not true // false not false // true not (5 == 6) // true`

Features

For loops

Catscript allows users to loop through the same chunk of embedded code with for loops. It might be more accurate to think of them as for each loops, and it iterates through every item in a set.

Start by using the keyword "for" followed by parenthesis "()". Within the parenthesis there is an identifier, then a separating comma, then the expression. After the parenthesis, there are curly braces "{}" and all the code within them is run once for each item in the expression. * Think of *for (x in nums)*, the loop will iterate through the embedded code once for each item x in nums.

```
for (x in [1, 2, 4]) {
    print(x + " ")
}
// the printed output will be: 1 2 4
```

Print statemetns

Users can display text in Catscript using print statements.

By using the keywork "print" followed by parenthesis"()", the expression within the parenthesis "()" will be printed.

```
print("Hello World")
// the printed output will be: Hello World
```

Functions

A function allows a block of code be called elsewhere. Often functions perform a specific task, that can be called to run elsewhere in the program. This allows users to write resusable code, and increase program readability.

Function Declaration

To declare a function, you start with the function name then can optionally follow with the function's return type. This is followed by parenthesis "()" that contain the parameters. Then there are curly braces "{}" that contain the body of the function. This is what will be executed when the function is run. Within the body you will need a return statement. * Return details: Similarly to variables, functions can be implicitly or explicitly typed. Unlike variables, implicitly typed funcitons are set to type "void", which is an exclusive type for functions to say that there will be no returned value. * Parameter details: There can be as many parameters as you like. Each parameter is entered in the format variable name optionally followed a collen ":" and its explicit data-type. If there is more than one, then the parameters are separated by a comma.

```
 // Example of an implicitly typed function return that returns nothing (void)
printPlusHello(name : string) {
    print("Hello " + name + "!")
}

// This is an example of the format, but cannot run, since it doesn't have a valid type
//functionName : return-type () {
    // body
    // return (value) // return if the type is not void
//}

// Example of an explicitly typed function return that returns nothing (void)
printPlusHi : void (name : string) {
    print("Hi " + name + "!")
}

// Example of an explicitly typed function return that returns an int
convertCelsuisToFeriheit : int (Celsuis : int) {
    int ret = ...
    return (ret)
}
```

Function Returns

Functions can return a single item. * The type of the return is defined in the function declaration. * When a return statement is reached, the function is exited. * There must always be at least one return statement that is always reached. * There can be multiple return statements in a function. * No return statement is needed if the data-type is void. But one without a value can still be used to exit the function early

```
// All branches are covered with a return statement
functionName : string (x : bool) {
  if (x) {
      return ("x is true")
  } else {
      return ("x is false")
  }
}

// This will have a parser error, since there is no return if x is false.
functionName : string (x : bool) {
  if (x) {
      return ("x is true")
  }
}

// Function won't print hello because it returns before reaching the print statement
myfunc () {
    return ()
    print ("Hello")
}
```

Function Calls

To access a function, you must call it. This can be done by typing the function name followed by parenthesis "()" containing all parameters that the function requires, separated by commas. If the function returns something, then it can be stored in a variable or used in an expression. The function call must have the same number of arguments as the function definition had parameters.

```
function functionName() {}

functionName()  // function is called



function convertCelsuisToFeriheit : int (temp : int) {
    return (temp * 9 / 5 + 32)
}

var farenheit : int = convertCelsuisToFeriheit(0)
print(farenheit) // function is called, and the returned 32 is printed
```

Conditional statements

Catscript allows users to make conditional statements using if statements. When the given expression evaluates to true, the embedded section of code is executed.

- If statements
    - In catscript if statements start with the keyword "if" followed by parenthesis "()". Within the parenthesis "()" there is an expression that evaluates to a boolean. This is followed by curly braces "{}" in which all code embedded within the curly braces is executed only if the initial expression evaluates to true.
    - `if (expression) { // all code embedded will be run only if the expression evaluated to true. print(x) }`

    The if statement

- Else If statements
    - An else if statement must be put after either an if statement or another else if statement.
    - This allows another expression to be checked only if the first expression evaluated to false.
    - After the previous conditional's closing curly brace "}" the keywords "else if" must be used followed by a set of parenthesis "()" containing an expression that evaluates to a boolean. This is followed by curly braces "{}" in which all code embedded within the curly braces is executed only if the *else if* expression evaluates to true.
    - `if (expression1) { print(x) } else if (expression2) { // all code embedded will be run only if expression1 evaluated to false and expression2 evaluated to true. print(x) }`

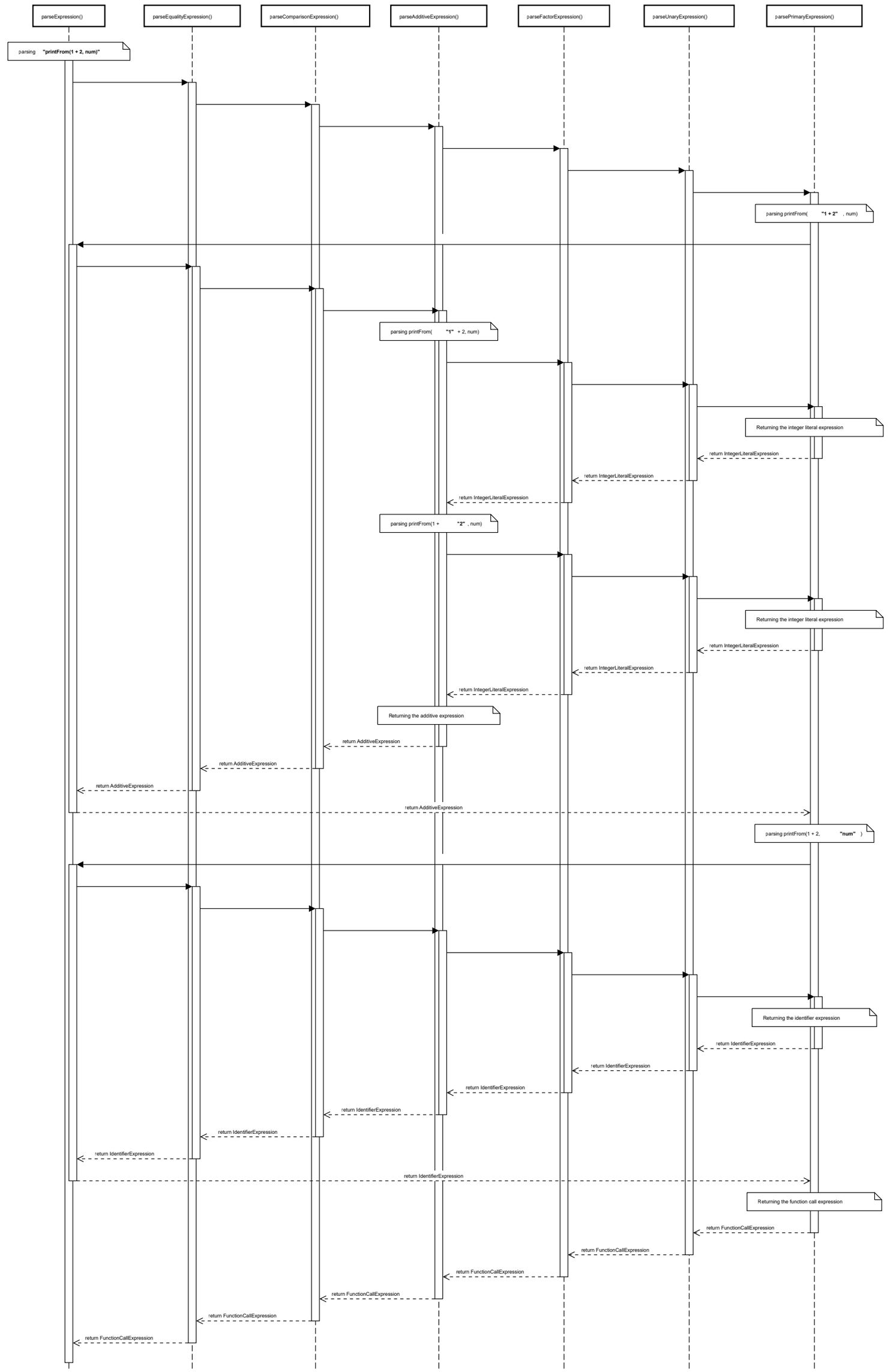
- Else statements
    - An else statement must be put after either an if statement or another else if statement.
    - After the previous conditional's closing curly brace "}" the keywords "else" must be used followed by curly braces "{}" in which all code embedded within the curly braces is executed only if the *else if* expression evaluates to true.
    - ``` if (expression1) { print(x) } else if (expression2) { print(x) } else { // all code embedded will be run only if expression1 and expression2 evaluated to false. print(x) }

  if (expression3) { print(x) } else { // all code embedded will be run only if expression3 evaluated to false. print(x) } ```

Section 5: UML

My UML diagram is a sequence diagram modeling the parsing of the expression “printFrom(1 + 2, num)”. I chose a sequence diagram because I wanted to see how functions accessed and interacted with each other as they parsed an expression. In this case, I chose a function call containing both an additive expression and an identifier expression. This unique combination causes a recursive call for the two expressions in the printFrom function call, returning both regular and literal expressions.

Parsing “printFrom(1 + 2, num)”



Section 6: Design trade-offs

In terms of recursive descent versus parser generators for compilers, there are clear advantages and disadvantages for both. Despite these differences, both work very similarly functionality wise: starting by generating tokens, parsing the tokens, creating a parse tree, evaluating the parse tree, then compiling into byte code. For recursive descent, it is much closer to the process of compilation in terms of how it is written. Each part of the compiler must be written in code by hand. Also, as its namesake suggests, it operates recursively to perform the different stages of compilation, which closely reflects the inherently recursive nature of coding grammars. This teaches more about the process of compilation through the writing of the compiler: breaking each step down into understandable chunks. The code is also easy to debug and understand: being significantly shorter, having clear variable and function names, and having comments. Due to these benefits, recursive descent is the method used in industry. However, this methodology is a double-edged sword, as having to write everything by hand means working at a low level in many cases. Recursive descent also requires more infrastructure upfront to make sure each piece is working properly. Overall, recursive descent is a more difficult approach than parser generators, but it creates a clearer understanding of compilers with a better end result.

As for parser generators, they are more frequently used in education as opposed to industry. They rely on lexical grammar and language grammar to generate a parser. This expedites the process of writing a compiler since it needs less handwritten code and doesn't require up-front infrastructure like recursive descent does. It's a much simpler process that is shaped by the definition of file configurations and logic, similar to what can be seen with BNFs and EBNFs. On the other hand, there is a reason why parser generators aren't used in industry. First, parser generators are inherently more declarative because of their creation process, which makes their underlying systems more abstract and unclear. Second, there are many parts of parser generators that are not directly related to compilers, but instead tie into learning a specific parser generator or BNF concepts. This causes parser generators to stray further from their inherent purpose and functionality. Third, code made by parser generators is near impossible to debug or change. Parser generator code is much longer, contains arbitrarily named variables and methods, and lacks any guidance in terms of comments. While parser generators are used frequently in academia, they are poor choices for creating a compiler because of how much they abstract the compilation process and the obfuscated results they produce.

Section 7: Software development life cycle model

We used Test Driven Development (TDD) for this project, which is heavily based on passing tests. This model has both been very useful and somewhat half-baked for the development of this project. In terms of positives, TDD makes debugging code extremely easy since it focuses on the results of different functions. It makes it immediately clear if something is not working properly, and shows with the result in which ways it is off. This can easily act as a springboard for debugging a project, since its clear what part of the program is failing. In addition, it provides guidance for what the result of method should be in the first place. This can act as a nice starting point for development, drawing a clear through line of what a certain method should accomplish.

On the other hand, TDD can also be finicky. Without proper tests, TDD can easily miss edge cases that may be important or even necessary for other parts of the project. This occurred multiple times throughout the development of this capstone project for me, since object type and location setting was not always checked and accounted for in the current testing suite. This would sometimes require major reworks of parts of the project to allow coherence of standards between portions. In addition, this segmentation of TDD generally makes the testing of software interactions harder. While this would still be an issue with other development systems, it remains a problem in TDD.

Overall, I'm glad that TDD was used for the development of this capstone. While it certainly has a couple of small drawbacks, the problems it has are shared by most other development strategies. In addition, the ease of debugging and clear goals that are set up by TDD make progress faster and simpler. Plus, most issues with TDD can be helped or even completely fixed by simply writing more complex tests to check more extreme edge cases.