Section 1: Program

Please include a zip file of the final repository in this directory.

Section 2: Teamwork

While I was responsible for the implementation of this project, my partner, Matthew Rolls, provided me with three tests which tested the functionality of my CatScript parser, and I did the same for him. My partner also created the design document for the CatScript language located in Section 4 of this document, and just like with the tests, I also provided him with a design document. Together we helped each other fulfill the requirements of this capstone project.

Section 3: Design pattern

One design pattern we implemented in our CatScript parser is the Memoization design pattern. This design pattern was implemented as a way to save memory. When parsing List Literal expressions during recursive decent, an instance of a `ListLiteralExpression` is created in the parse tree. This `ListLiteralExpression` instance has a `ListType` instance associated with it defining the type of the the `ListLiteralExpression`. When assigning a `ListType` to a `ListLiteralExpression` a method is called on line 37 of the `CatscriptType` class. This method, `getListType(CatscriptType type)` takes a `CatscriptType` such as `CatscriptType.INT` and returns a `ListType` of the same type. List Types are similar to regular types but they are unique to lists. Every time the `getListType(CatscriptType type)` method is called, it creates a new `ListType` instance of the given `CatscriptType` if one hasn't been created already. These `ListTypes` are then stored in a hashmap along with their associated `CatscriptType` such that a new instance of `ListType` is not created every time. This type of memoizing keeps redundant `ListType` instances out of memory.

```
37  @   public static CatscriptType getListType(CatscriptType type) {
38          if (typeMap == null) {
39              typeMap = new HashMap<>();
40          }
41          ListType returnType = typeMap.get(type);
42          if (returnType == null) {
43              returnType = new ListType(type);
44              typeMap.put(type, returnType);
45          }
46          return returnType;
47      }
48
```

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

Catscript Documentation

Introduction

Catscript is a simple scripting language that consists of program statements which can be either statements or function declarations. A function declaration is a program statement that consists of a function name, a parameter list, an optional return type and any number of function body statements. Function body statements can be either a standard statement or a return statement. Declaring a function allows for a block of code to be called by function call statements that correspond to it. A statement can be a function call statement, a for statement, an if statement, a print statement, a variable statement or an assignment statement. These statements consist of expressions and/or additional statements to form the features of the Catscript language. Catscript at its lowest level is formed by its types which are 32 bit integer, string, boolean, list, null and object.

Features

For Loop

The For Loop is defined by the for statement which requires the key word "for" and an identifier such as a variable name and an expression contained in parentheses. The body of the for loop contained in braces contains any number of statements. The identifier and expression define how the loop should iterate and the statements define what the loop will execute each time it iterates.

```
for ( x in 5 ) {
        x = x + x;
}
```

If Statement

The If statement requires the keyword "if", an expression and any number of statements contained in its braces. Optional sections of the if statement includes any number of else if statements and its associated any number of statements and an optional single else statements and its associated any number of statements. The control flow of the if statement is simple. The program starts at the initial if statement and if the expression evaluates to true the statements contained in its braces will execute and then once completed the program will break out of the loop. If the statement evaluates to false the program will look at the next statement if it exists to see if that statement will evaluate to true and execute its statements before breaking out of the statement. An else statement will always evalute to true if reached in the control flow.

```
if ( a == 0 ) {
        x = x + 1;
} else if ( a == 1 ) {
        x = x + 2;
} else {
        x = x + 3;
}
```

Print Statement

The Print statement requires the keyword "print" and only an expression followed by a semicolon. Upon executing the print statement will "print" the expression. Typical implementations may have the expression in the form of a string.

```
print("Hello World");
```

Variable Statement

The Variable statement requires the keyword "var", an identifier to be the name of the variable and an expression to assign to the variable followed by a semicolon. Optionally, between the name and the assignment an explicit type may be specified by including a colon and a type expression. This statement will functionally declare a variable an assign it a value.

```
var y : int = 20;
```

Function Call Statement

The Function call statement requires only a function call expression followed by a semicolon. Functionally this statement just calls a declared function and the code of the function is executed.

```
funca ( x );
```

Assignment Statement

The Assignment statement is incredibly similar to the variable statement except it requires a variable that has already been declared. The assignment statement is only assigning a value to the preexisting variable. It takes the form of an identifier, assignment "=" and an expression to assign to the variable and a trailing semicolon.

```
name = "Jeff";
```

Function Declaration Statement and Function Body Statement

The Function declaration statement requires the key word "function", an identifier, a parameter list contained in parentheses and an optional specified return type which consists of a colon and type expression. Additionally, within its braces is contained any number of function body statements which can be either a statement or return statement. A function declarations declares a function that can be called by function call statements causing the function's statements to execute using an optional parameters and returns.

```
function funca ( x ) : int {
        x = x + 2;
        return x;
}
```

Parameter List and Parameter

The Parameter list statement can be empty or requires any number of parameters separated by commas.

```
x : int, y, name : string
```

The parameter requires an identifier and an optional colon and type expression to specify a type. A parameter is functionally just a variable.

```
x
```

or

```
x : int
```

Return Statement

The Return statement requires the keyword "return" and an optional expression followed by a semicolon. This statement allows for a value to be returned to where a function was called.

```
return x;
```

or

```
return x + 1;
```

Expressions

Equality Expression

The Equality expression requires a left hand side expression and a right hand side expression with either a "!=" or "==" between them as an operator. A not equals will mean the expression will return true if the expressions are not equal and an equals equals will mean the expression returns true if the expressions are equal.

```
x == y
z != a
```

Comparison Expression

The Comparison expression acts similarly to the equality expression and requires a left hand side expression and a right hand side expression with either a ">", ">=", "<" or "<=" between them as an operator. The expression will return true if whichever of the specified operators is a true description of it compared to the left side.

```
10 >= 5
```

Evaluates to true

```
10 <= 5
```

Evaluates to false

Additive Expression

The Additive expression consists of a left hand side expression and a right hand expression side split by either a "+" or "-" operator. The expression will simply add or subtract the two expressions.

```
5 + 5
5 - 5
```

Factor Expression

The Factor expression acts similarly to the additive expression and onsists of a left hand side expression and a right hand expression side expect they are split by a "/" or "*" operator. The expression will simply multiply or divide the expressions.

```
5 / 5
5 * 5
```

Unary Expression

The Unary expression consists of a "not" or "-" operator and a right hand side of either a unary expression or primary expression. This expression allows for negative numbers using "-" and not functionality for booleans using "not".

```
not a
not not a
-5
```

Primary Expression

The primary expression takes the form of either an identifier, string, integer, "true", "false", "null", list literal, function call or any number of parenthesized expressions.

```
5
Hello
true
null
```

List Literal

The List literal requires to brackets to contain at least one expression with the option for any number of additional expressions separated by commas. A list literal can also contain list literals.

```
[5, 6, 7]
["Hello", "Howdy"]
```

Function Call

The Function call requires an identifier that corresponds to a function and an argument list contained in parentheses. The function call calls the corresponding function and passes the arguments to it for execution of the function.

```
funca ( x )
```

Argument List

The Argument list consists of nothing or an expression or any number of expressions separated by commas. As the argument list uses expressions the arguments can take multiple forms such as raw values or variables.

```
a, b, c, d
a, 5, c, "Hello"
```

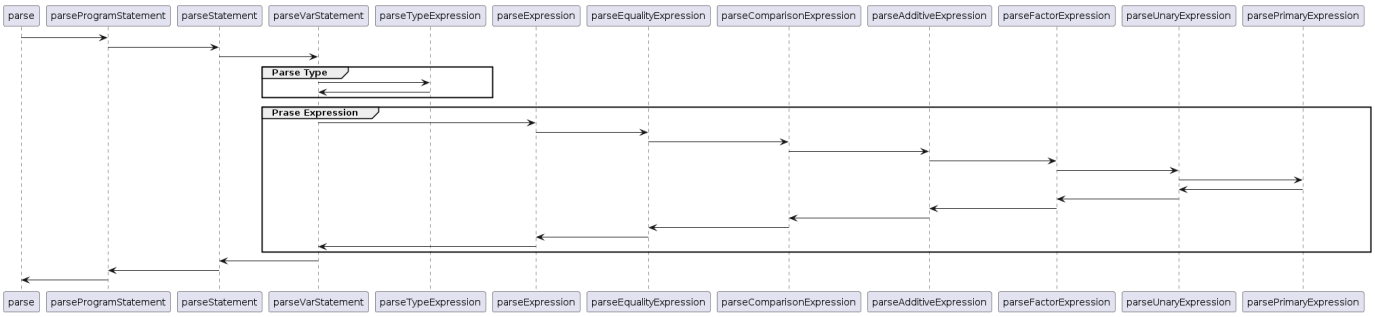
Type Expression

Type expressions are expressions indicating type. These include 'int', 'string', 'bool', 'object' and 'list' or 'list' followed by any number of type expressions contained within '<' and '>'.

```
list < int >
list < int > < int >
```

Section 5: UML.

The following UML diagram depicts the recursive decent parsing of `var x = 1 ;`



Section 6: Design trade-offs

There are two main options when designing a parser: Recursive Decent and Parser Generators. Recursive Decent is an algorithm in which each parse element is parsed recursively. This essentially boils down to being an elaborate Depth First Search algorithm. Recursive decent algorithms are handwritten like in the case of this project and give the designer complete control over the parsing process.

Parser Generators are exactly as the name describes, in that they generate parsers. Usually, to generate a parser, a grammar for the language to be parsed is given to the parser generator, which in turn generates a parser. To user the parser, the designer will still need to implement a tokenizer to generate a token stream which will then be passed to the parser for parsing

The major advantage of writing a Recursive Decent algorithm rather than using a Parser Generator is that, as mentioned, the designer has complete control over the behavior of the parser as well as being able to not so strictly adhere to a grammar. This also allows for potentially higher quality code as generated parsers can often include unnecessary or poorly generated code and can also produce unexpected parsing behaviors.

The major advantage of using a Parser Generator is that it is quick and easy, and that's about it.

For this project, we chose to write our own Recursive Decent parsers in order to better understand how parsing works. Additionally, the majority of production parsers also use Recursive Decent, so this experience of writing a Recursive Decent parser actually translates to the real world.

Section 7: Software development life cycle model

For this project we used Test Driven Development (TDD). TDD works by writing a comprehensive set of tests to thoroughly verify the functionality of the software system. For this project, we grouped the tests into checkpoints for different parts of the system such as the tokenizer, parser, and compiler. This style of development was beneficial to our team in that by tackling each component of the software system one at a

time through testing and verifying that there were little to no bugs, it made debugging future components of the system much easier.