Catscript Compiler

CSCI 468 - Compilers

Capstone Project Portfolio

Bryce Leighton

Simeon Shirshov

Montana State University

Spring 2023

Section 1: Program

The source code for this project is included in this directory titled source.zip.

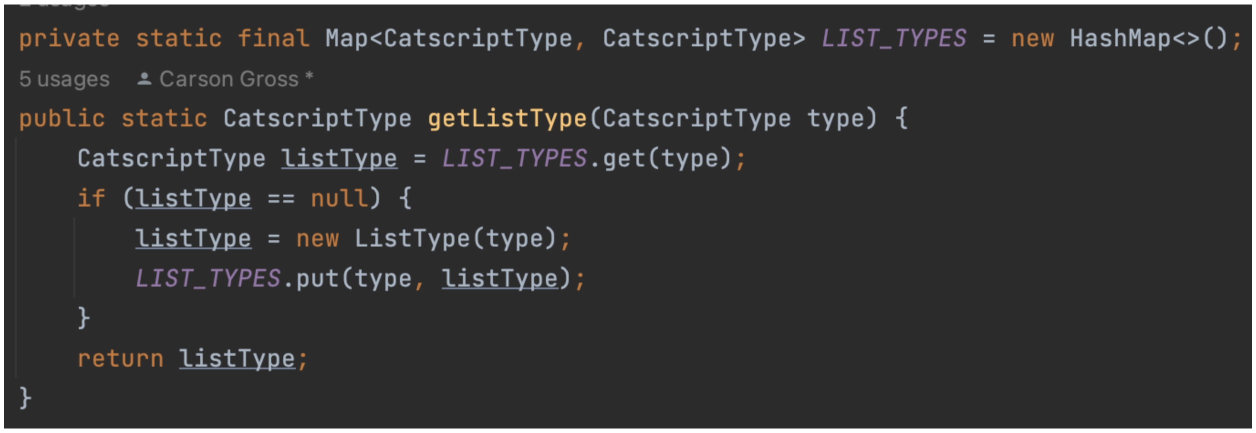
Section 2: Teamwork

This capstone project was completed with a team of 2 people, team member 1 (Simeon Shirshov) and team member 2 (Bryce Leighton). The completion of this project was broken up into 4 different independent steps and an additional 5th step which was collaborative. For the first 4 steps, Tokenization, Expression Parsing, Statement Parsing + Eval, and Compilation of Catscript, team members 2 and 1 independently developed the code using test-driven development with identical test suites. For the last step, additional cumulative testing, team members 2 and 1 created new tests and traded them with each other. In addition to that aspect of teamwork, teamwork was also highly prevalent when creating the documentation for the structure of Catscript seen in section 4. Team member 2 wrote all the Catscript documentation for this project, team member 1 wrote all the documentation for team member 2's project. The workload for this project between the two members was about 90% for team member 1 and 10% for team member 2. This was the opposite for team member 2's project. For this project team member 1 was the primary engineer and team member 2 was the documentation and test engineer.

Partner tests file path: **src/test/java/edu/montana/csci/csci468/demo/Scratch.java**

Section 3: Design Pattern

An example of a design pattern used within the Catscript parser is the Memoization Pattern also sometimes called the Flyweight Pattern. The idea with this design choice is to minimize memory utilization through the use of sharing. The way this is done can be seen below with the snippet of code pulled from the Catscript compiler. Essentially what the code is doing is avoiding creating multiple identical ListType objects and instead using a single ListType object that is referenced before a new type is added. The end result is that the memory footprint of our application is reduced and the performance improved.



```java
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
5 usages   Carson Gross *
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);
    if (listType == null) {
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
    return listType;
}
```

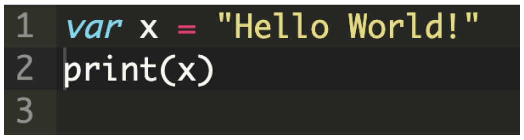
File path: src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java

Section 4: Catscript Documentation

Introduction:

Catscript is a simple programming language with syntax similar to that of Python and Java. Catscript is supportive of loops, functions, conditionals, dynamic typing, and a list data type. It is dictated by a simple language structure that is discussed below. An example is shown below of a simple Catscript "Hello World!" program, and further functionality is documented below.

'Hello World' Example



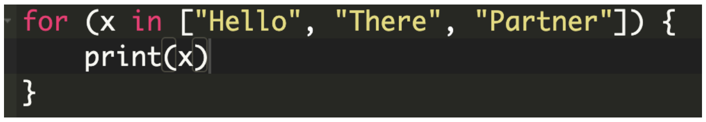
```
var x = "Hello World!"
print(x)

```

Statements:

For Loop Statement:

Catscript For Loops statements iterate over elements in a list and are able to execute code for each element with a loop variable by using an identifier and an expression. They are able to execute any statements inside of the For Loop, which allows for nesting of loops.

For Loop Example:

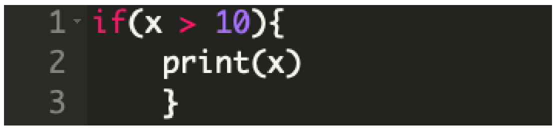


```
for (x in ["Hello", "There", "Partner"]) {
    print(x)
}
```

If Statement:

If Statements evaluate a certain condition to be true or false. If the condition is true, then the code within the If Statement will be executed. An Else Statement after an If Statement is an option, and the code within an Else Statement will be executed if the condition is false.

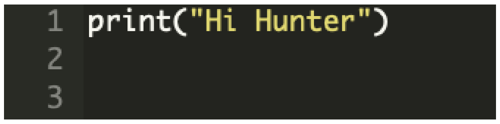
If Statement Example:



```
if(x > 10){
    print(x)
    }
```

Print Statement:
Print Statements in Catscript are used to output the value of an expression to the console. Catscript will also evaluate any expressions that are placed inside of a print statement and print the result of the evaluation.

Print Statement Example:

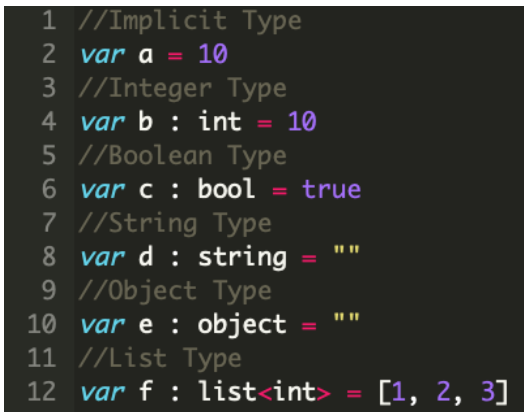


```
print("Hi Hunter")
```

Variable Statement:

Variable Statement are used to declare and optionally assign a value to a variable. There is also an option to annotate its type. Variables can be both implicitly and explicitly typed.

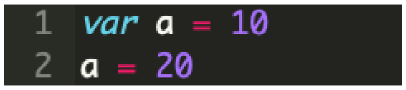
Variable Statement/Catscript Types Example:



```
//Implicit Type
var a = 10
//Integer Type
var b : int = 10
//Boolean Type
var c : bool = true
//String Type
var d : string = ""
//Object Type
var e : object = ""
//List Type
var f : list<int> = [1, 2, 3]
```

Assignment Statement:
Assignment Statements in Catscript are used to assign a new value to an existing variable. One important stipulation is that Catscript only allows reassignment of a variable if the new assignment value is of the same type as the previous assignment. This protects the type of the current variable. As shown below, an integer variable can only be assigned integer values.

Assignment Statement Example:

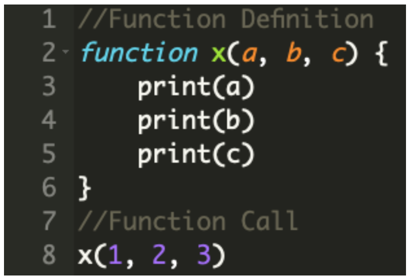


```
var a = 10
a = 20
```

Function Call Statement:
The Function Call Statement in Catscript is used to call a predefined function with a set of arguments if applicable. A function call statement requires that all required arguments are given so that the function has all of its needed parameters.

Function Declaration/Definition Statement:
Function Declaration/Definition Statement in Catscript is used to define a new function. The Statement requires a function name and a block of code (body) within the function which the function executes. Optionally Function Declaration Statements also allow for a list of parameters as input which can be used within the function and can also have return type annotation. Function Definition Statements are able to have additional statements nested within the body.

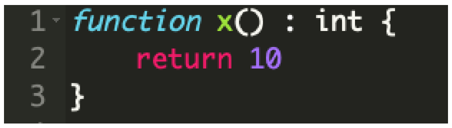
Function Definition/Call Example:



```
//Function Definition
function x(a, b, c) {
    print(a)
    print(b)
    print(c)
}
//Function Call
x(1, 2, 3)
```

Return Statement:
The Return Statement in Catscript is used to return a value from a function. Additionally Return Statements complete/break when hit. Return Statements can be assigned any type of expression, because when the Return Statement is reached it will evaluate that expression.

Return Statement Example:

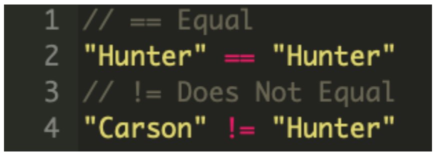


```
function x() : int {
    return 10
}
```

<u>Expressions:</u>

Equality Expression:
The Equality Expression in Catscript can either have a bang equal (!=) or equal equal (==) which split two different expressions to be evaluated as an Equality Expression. Equality Expression’s return a Boolean value of True or False. The Equal Equal returns True when expressions on either side are equal. Bang Equal returns True when expressions on either side are not equal.

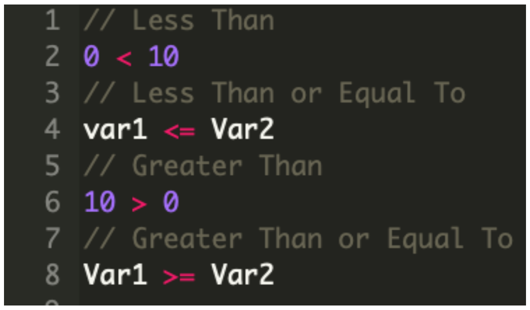
Equality Expression Example:



```
// == Equal
"Hunter" == "Hunter"
// != Does Not Equal
"Carson" != "Hunter"
```

Comparison Expression:
The Comparison Expression in Catscript have similar logic to the Equality Expressions. In Catscript, Comparison Expressions are used to compare two values. They can use the ">" , ">=", "<", or "<=" operators, which are the greater, greater than or equal to, less than, and less than or equal to operators respectively. When the operator used is true for the two expressions on either side Catscript returns True. When it’s false it returns False.

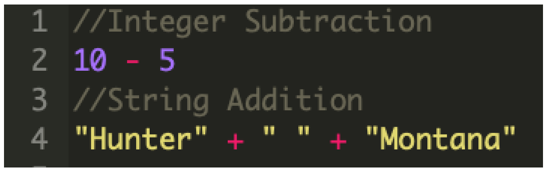
Comparison Expression Example:



```
// Less Than
0 < 10
// Less Than or Equal To
var1 <= Var2
// Greater Than
10 > 0
// Greater Than or Equal To
Var1 >= Var2
```

Additive Expression:
The Additive Expression in Catscript is used to perform addition or subtraction on two values. It can use the addition "+" or subtraction "-" operator. In Catscript the values are evaluated from left to right. The Additive Expression also supports string concatenation and can be used to concatenate strings together with the “+” symbol.

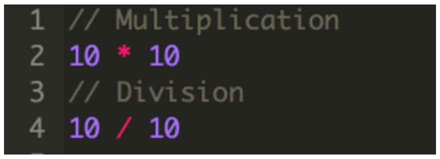
Additive Expression Example:



```
//Integer Subtraction
10 - 5
//String Addition
"Hunter" + " " + "Montana"
```

Factor Expression:
The Factor Expression in Catscript is used to perform multiplication or division on two values. It can use the "*" or "/" operator. The star “*” operator is used for multiplication. The slash “/” operator is used for division. Factor Expressions are similar to Additive expressions in that they are evaluated from left to right.

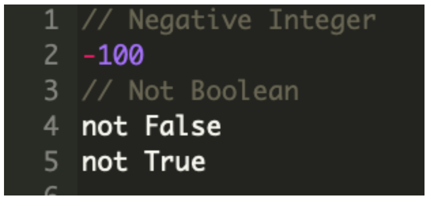
Factor Expression Example:



```
// Multiplication
10 * 10
// Division
10 / 10
```

Unary Expression:
The Unary Expression in Catscript is used to apply a unary operator to an expression. It can use the "not" or "-" operator. Generally the “not” operator is used to negate a value such as a boolean, int, or string. The “-” operator is used to symbolize a negative integer. The “-” operator can only be used with integers.

Unary Expression Example:



```
// Negative Integer
-100
// Not Boolean
not False
not True
```

Primary Expression:
The primary expression in Catscript is a basic type of expression that produces a value. It can be an Identifier/variable, a string, an integer, a boolean, a null value, a list literal, or a function call.

Identifier/Variable: An identifier or a variable in Catscript is a string chosen by the user to represent something, like a variable name.

String: A string in Catscript is a java-style string.

Integer: An integer in Catscript is a 32 bit integer.

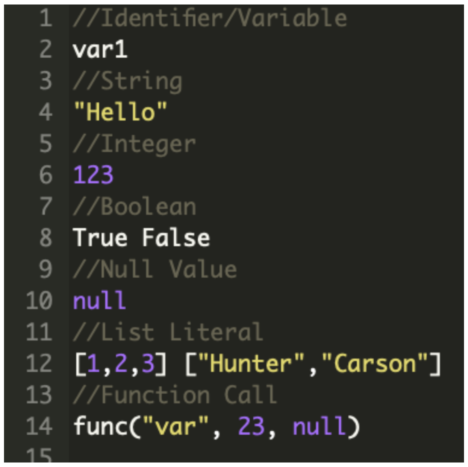
Boolean: A boolean in Catscript is an expression used to represent True or False Values.

Null Value: A null value in Catscript is an expression used to show an absence of value.

List Literal: A list literal in Catscript is used to create a new list object. It takes a series of expressions as values.

Function Call: A function call in Catscript is used to call a function with a set of arguments.

Primary Expression Examples:



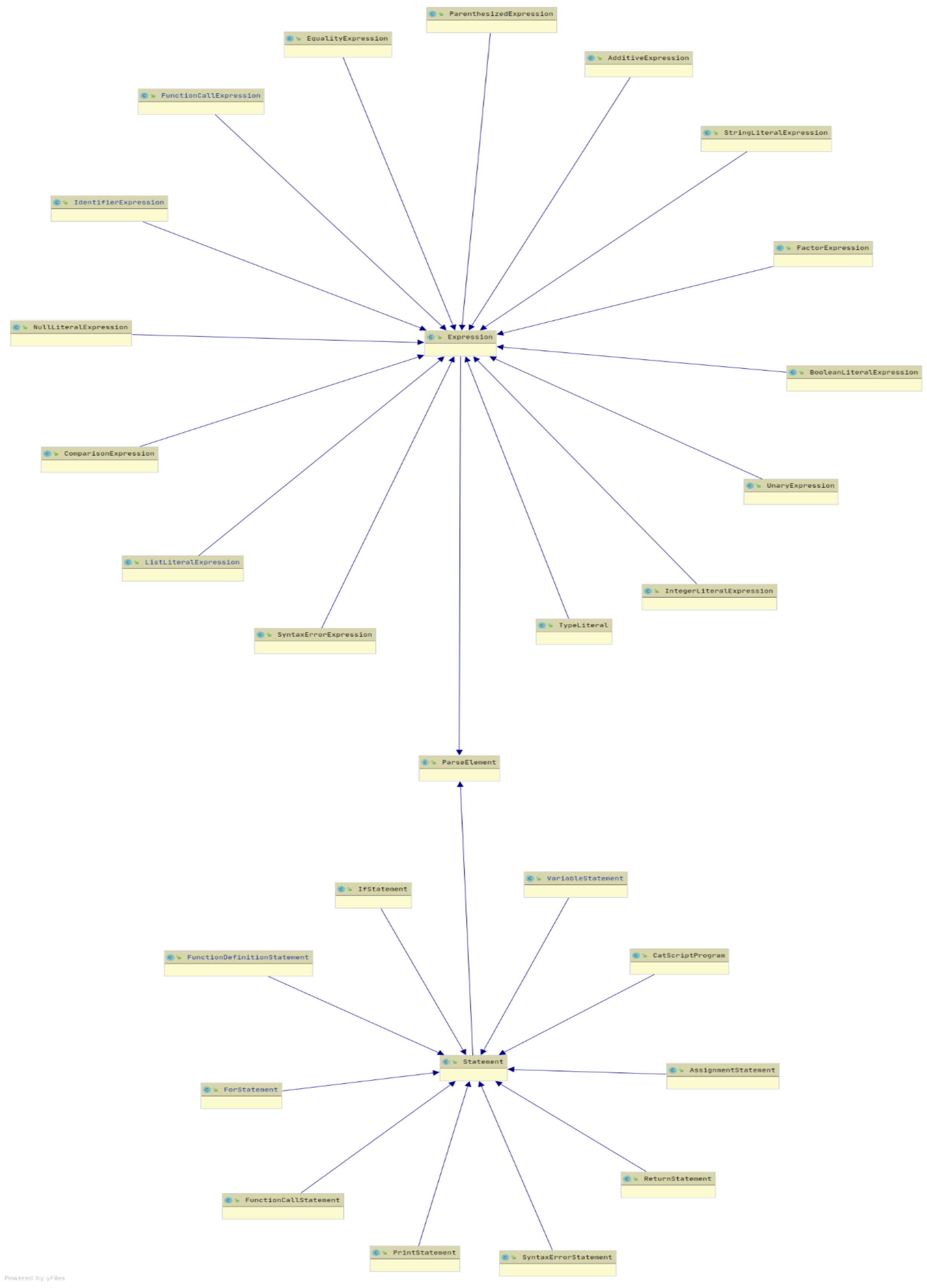
```
//Identifier/Variable
var1
//String
"Hello"
//Integer
123
//Boolean
True False
//Null Value
null
//List Literal
[1,2,3] ["Hunter","Carson"]
//Function Call
func("var", 23, null)
```

Type Expression:
A type expression in Catscript is used to annotate a variable or a function parameter with a data type. It can be "int", "string", "bool", "object", or "list" with an optional generic type annotation.

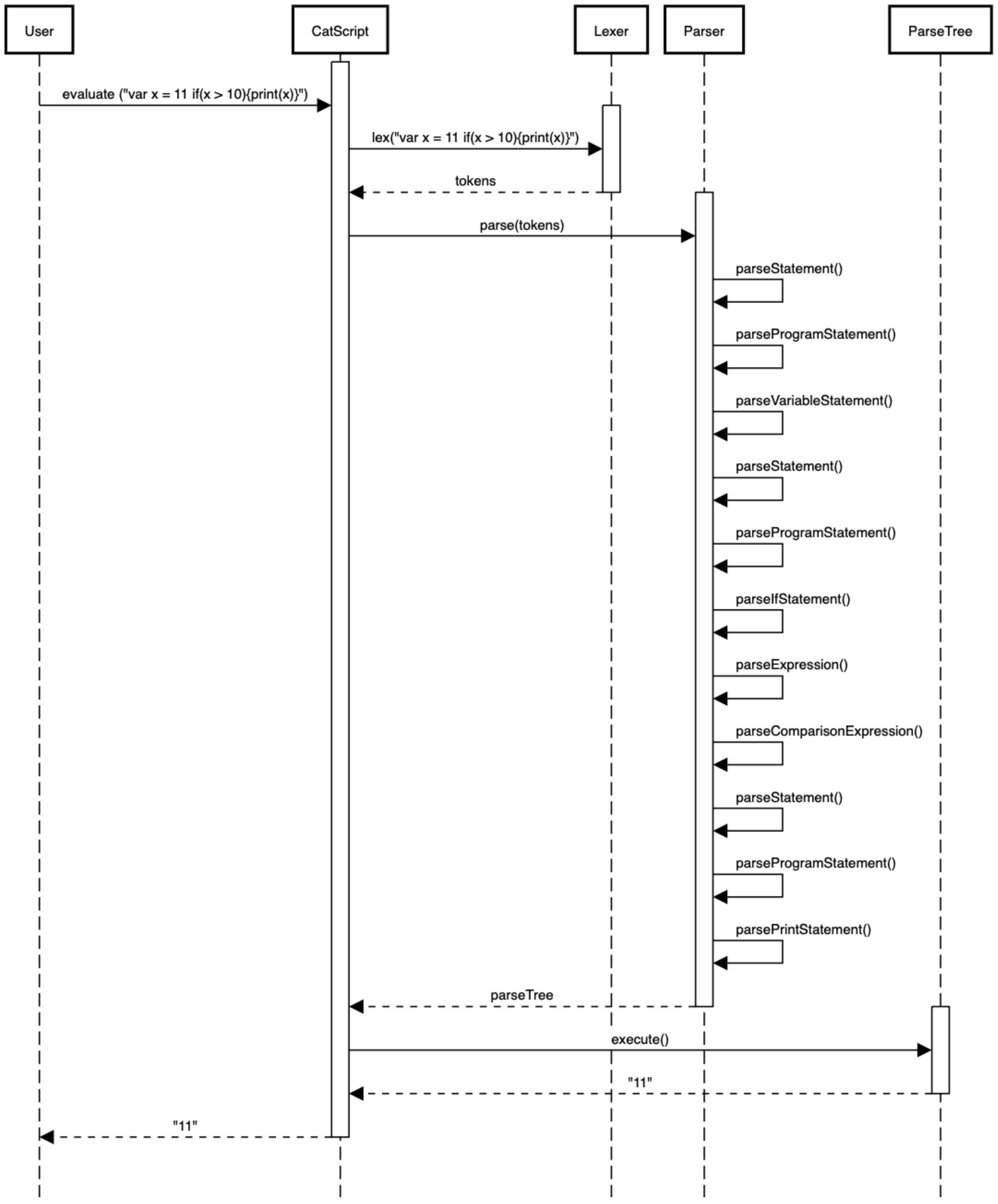
```
catscript_program = { program_statement };

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, '>']
```

Section 5: UML

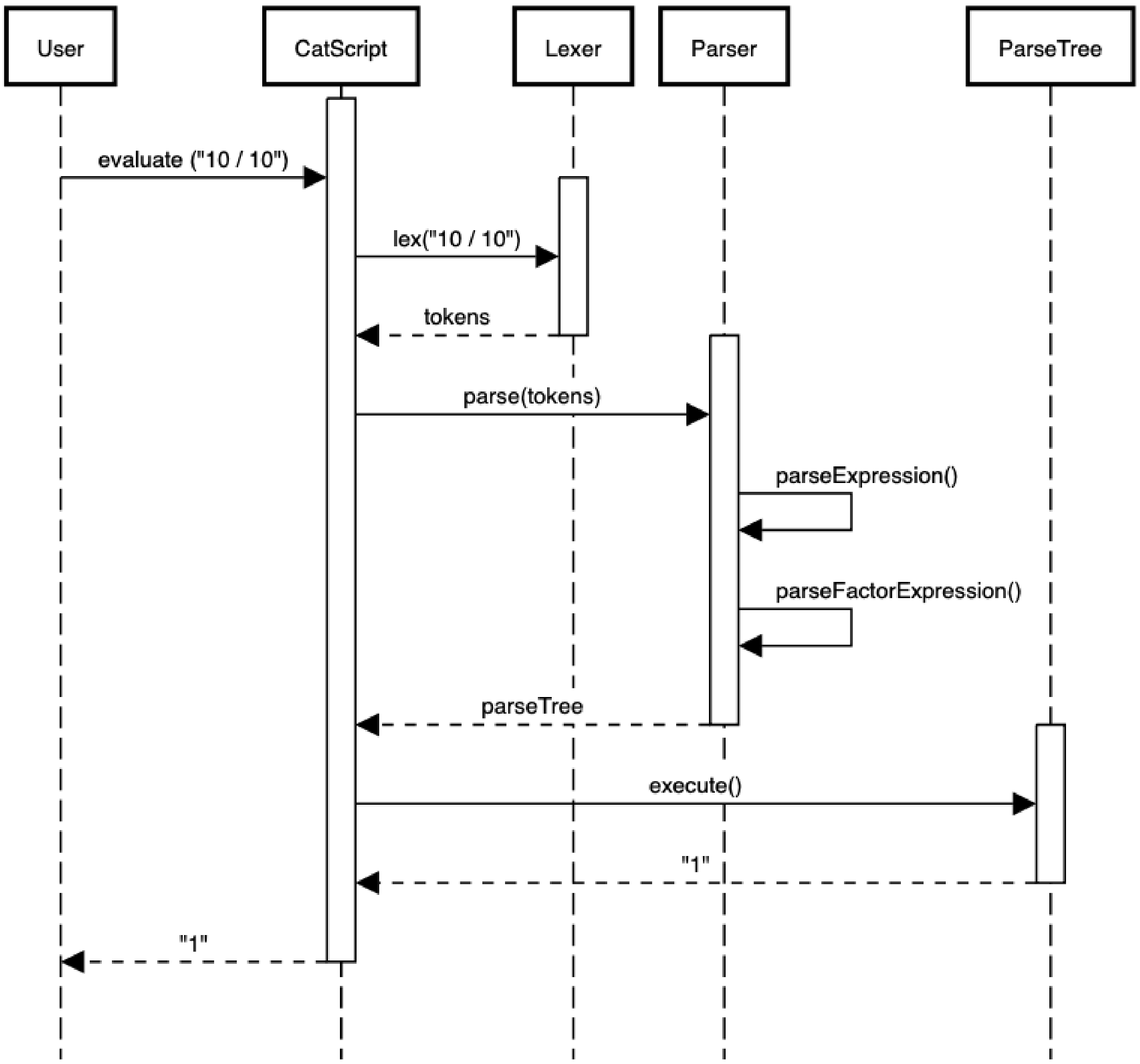
The first UML Diagram below shows a general overview of the general structure of Catscript. The couple Sequence Diagrams after that help demonstrate the functionality of the Catscript compiler and the program sequence that happens in the background when compiling certain code.


ParenthesizedExpression
EqualityExpression
AdditiveExpression
FunctionCallExpression
StringLiteralExpression
IdentifierExpression
FactorExpression
NullLiteralExpression
Expression
BooleanLiteralExpression
ComparisonExpression
UnaryExpression
ListLiteralExpression
IntegerLiteralExpression
SyntaxErrorExpression
TypeLiteral
ParseElement
IfStatement
VariableStatement
FunctionDefinitionStatement
CatScriptProgram
Statement
AssignmentStatement
ForStatement
ReturnStatement
FunctionCallStatement
PrintStatement
SyntaxErrorStatement
Powered by yFiles

Catscript If With Print Statement Sequence Diagram



Catscript Division Sequence Diagram



Section 6: Design Trade-Offs

The major design trade-off we decided to take with the process of developing a compiler for Catscript was creating the parser by hand rather than building it using a parser generator. Although using a parser generator would be much easier and likely require much less code, developing the parser by hand has a few different beneficial aspects to it. The primary benefit came from the way we built the parser which was using a recursive descent algorithm. This approach helped us gain crucial and complete understanding of exactly how grammars work that we otherwise wouldn’t have. Thanks to that key reason among other smaller reasons we believe that this design trade-off was justified.

Section 7: Software Development Life Cycle Model

Test-driven development was the software development life cycle model used for the development of this project. This method of software development consists of the creation of sets of tests for each portion of the software being developed and then the use of those tests to guide the actual code-writing process. In the case of the Catscript compiler, the tests were split into 4 different groups: tokenization, expression parsing, statement parsing + eval, and compilation. Each of the tests was designed to not only comprehensively cover edge cases but also build on top of each other. This thought-out design helped the code-writing process in a multitude of different ways, the most prevalent being the debugging of code. The tests allowed for easy identification of code that was necessary to implement and made it easy to see exactly where the issue lies if a test failed. Overall we believe this was the best approach to the development of this project because of the way it helped us maintain efficiency by providing clear goals and a much more clear understanding of the functionality we were building.