

# Capstone Portfolio

CSCI 468: Compilers  
Spring 2022

Team members:  
Alonso Darias  
Miles Naberhaus

## Section 1: Source Code

A link to the source code for the Catscript compiler:

<https://github.com/addariasfr/csci-468-spring2022-private/blob/master/capstone/portfolio/source.zip>

## Section 2: Teamwork

For this project I collaborated with Miles Naberhaus. We worked together to write the recursive descent parser for Catscript and a number of tests for the language. I, Alonso Darias, was the main programmer. I wrote the source code and most of the report excluding the Technical Documentation. Miles, who served as the tester and documenter for this project, wrote the technical documentation as well as three tests for my source code which are contained in the `src/test/java/edu.montana.csci468/demo/PartnerTest.java` file.

## Section 3: Design Pattern

The design pattern I utilized in this project is the memoization of the list type access method. This involves storing previously retrieved list type objects in a cache to save time and memory space when the same type is requested again. The second time a certain list type is requested, the value in cache can be returned rather than taking the steps to create a new type object.

## Section 4: Technical Documentation

### Introduction

Catscript is a simple, statically typed computer programming language built on top of Java. It implements recursive descent parsing such that the parse trees are easy for users to understand and use. There are two primary types of parse elements in Catscript, expressions and statements. Expressions evaluate to some value, while statements execute some computation. The typing system in Catscript includes integers, strings, booleans, lists, null, and objects, with many parse elements able to be explicitly given a type. The basic parse of Catscript begins with a `CatscriptProgram` statement, which then handles the parsed elements found during runtime. Below, we will give short descriptions of each parse element in the Catscript language, along with example Catscript code demonstrating their functionality and syntax.

### Parse Elements

#### *Expressions*

##### *Literals:*

##### **BooleanLiteralExpression**

In Catscript, a boolean value (true or false), can be stored as a `BooleanLiteralExpression`.

```
var bool = true
```

### IntegerLiteralExpression

Integers are represented (when using variables) with the Catscript `IntegerLiteralExpression` class. The Catscript below will evaluate to the integer value 1.

```
var intExample = 1
```

### ListLiteralExpression

Catscript represents lists as `ListLiteralExpressions`, which use brackets and commas to define the elements they contain. The below example specifies the explicit type of elements it will contain as strings.

```
var l : list<string> = ["one", "two", "three"]
```

### NullLiteralExpression

The value of null can be represented in Catscript as an instance of a `NullLiteralExpression`, which simply has the value null.

```
var nullExample = null
```

### StringLiteralExpression

A basic string is implemented in Catscript with the `StringLiteralExpression` class, which stores a string. As with other expressions, it can be explicitly or implicitly typed.

```
var implicit = "string one"  
var explicit : string = "string two"
```

### TypeLiteral

A `TypeLiteral` holds the value of one of the Catscript types, which are *INT*, *STRING*, *BOOLEAN*, *OBJECT*, *VOID*, and *NULL*. The below code shows a variable being assigned to a string value with the explicit type *STRING*.

```
var x : string = "string"
```

## Mathematical and Logical:

### AdditiveExpression

An `AdditiveExpression` is simply a Catscript expression that represents an additive

problem, with a left hand side, a right hand side, and an operator. Below are two AdditiveExpressions, with a mathematical operation and a string concatenation.

```
print(1 - 1)
print("string one" + "string two")
```

### FactorExpression

Multiplication and division are handled in Catscript FactorExpressions, containing a right hand side, left hand side, and an operator (similar to AdditiveExpression).

```
print(1 / 1)
print(10*3)
```

### ParenthesizedExpression

A ParenthesizedExpression is the addition of parentheses into some expressions to set their operation order.

```
print((1+4)*4)
```

### ComparisonExpression

The ComparisonExpression is used in Catscript to compare objects, with the four comparisons used as operators being greater than, less than, greater than or equal, and less than or equal. Like AdditiveExpression and FactorExpression, it has a right hand side and a left hand side. These expressions evaluate to boolean values.

```
print(1>2)
```

### EqualityExpression

The equality expression is the ComparisonExpression without the relational comparisons, as it only checks for object equality. The following two examples are comparing BooleanLiteralExpressions with the two logical equality operators. It also has a right hand side and a left hand side.

```
print(true==false)
print(true!=false)
```

### UnaryExpression

The UnaryExpression is the Catscript representation of the logical “not” and mathematical negative values.

```
var negOne = -1
var notTrue = not true
```

## ***Computing:***

### **FunctionCallExpression**

The FunctionCallExpression is a ParseElement in Catscript that contains the Identifier and arguments necessary to call a function. Functions are called by first referencing a Function's Identifier, and the passing its arguments in through parentheses.

```
foo(true)
```

### **IdentifierExpression**

An IdentifierExpression is essentially a name in Catscript, one which references some function or variable. Below, “bar” and “foo” are IdentifierExpressions.

```
var bar = "something"  
foo()
```

### **SyntaxErrorExpression**

Syntax errors are represented in Catscript as SyntaxErrorExpressions when they are parsed. Here is an example, where a list does not have an opening or closing bracket, resulting in its parse returning a SyntaxErrorExpression.

```
var errorList = 1, 2,
```

## ***Statements***

### ***Functions:***

#### **FunctionDefinitionStatement, FunctionCallStatement, and ReturnStatement**

A FunctionDefinitionStatement in Catscript requires syntax as follows: ‘function *Identifier* ( *ParameterList* ) { *Body* },’ where the *Identifier* is an IdentifierExpression, the *ParameterList* is a list of IdentifierExpressions (with optional explicit typing if followed by ‘: TypeLiteral’), and *Body* is some number of Statements with an optional ReturnStatement (‘return *Expression*’). A FunctionCallStatement is syntactically written ‘*Identifier* ( *arguments* ),’ where the *Identifier* is an IdentifierExpression that maps to some FunctionDefinitionStatement and *arguments* are a set of Expressions. Below is an example, in which the Function foo() is called with the argument “foo,” which it will print, and then return “bar,” which is then printed outside once it has returned. The major difference between FunctionCallStatement and FunctionCallExpression is that FunctionCallStatement executes, while FunctionCallExpression

evaluates (a core difference between all Statements and Expressions).

```
function foo(x : string) { print(x) return "bar" }  
print(foo("foo"))
```

## Computing:

### ForStatement

The Catscript ForStatement is syntactically generated with ‘for (*Identifier* in *Expression*) { *Statements* },’ where *Identifier* is the iterator we will be using, *Expression* is the Expression we will iterate over, and *Statements* are the set of Statements to be executed on each iteration of the loop. The example below will print out the values in the ListLiteralExpression given.

```
var iterList = [1, 2, 3, 4]  
for (i in iterList) { print(i) }
```

### IfStatement

The Catscript IfStatement is syntactically generated with ‘if ( *Expression* ) { *Statements* },’ followed by an optional ‘else ( IfStatement )’ or ‘else { *Statements* },’ *Expression* is some Expression that if evaluated to true will continue into the IfStatement, and *Statements* are the Statements to be conditionally executed.

```
if (false) {  
    print("I will not print")  
} else if (false) {  
    print("I will also not print")  
} else {  
    print("I will print")  
}
```

### PrintStatement

The PrintStatement, which we have been using to demonstrate code throughout this documentation, should need little explanation. Its syntax is simply ‘print ( *Expression* ),’ where *Expression* is some Expression whose value we would like to print.

```
print(true)  
print(4)  
print("String")  
var printList = [1, 2]  
print(printList)
```

### VariableStatement

Much like the Catscript PrintStatement, the VariableStatement's use should be somewhat apparent (it sets a variable's value to something or declares a variable). Its syntax is 'var *Identifier*' followed by an optional explicit type declaration ': type' and ending with '= *Expression*.' Here, *Identifier* is an IdentifierExpression that will be the new variable's name, and *Expression* is some Expression which will be evaluated to set the value of this new variable.

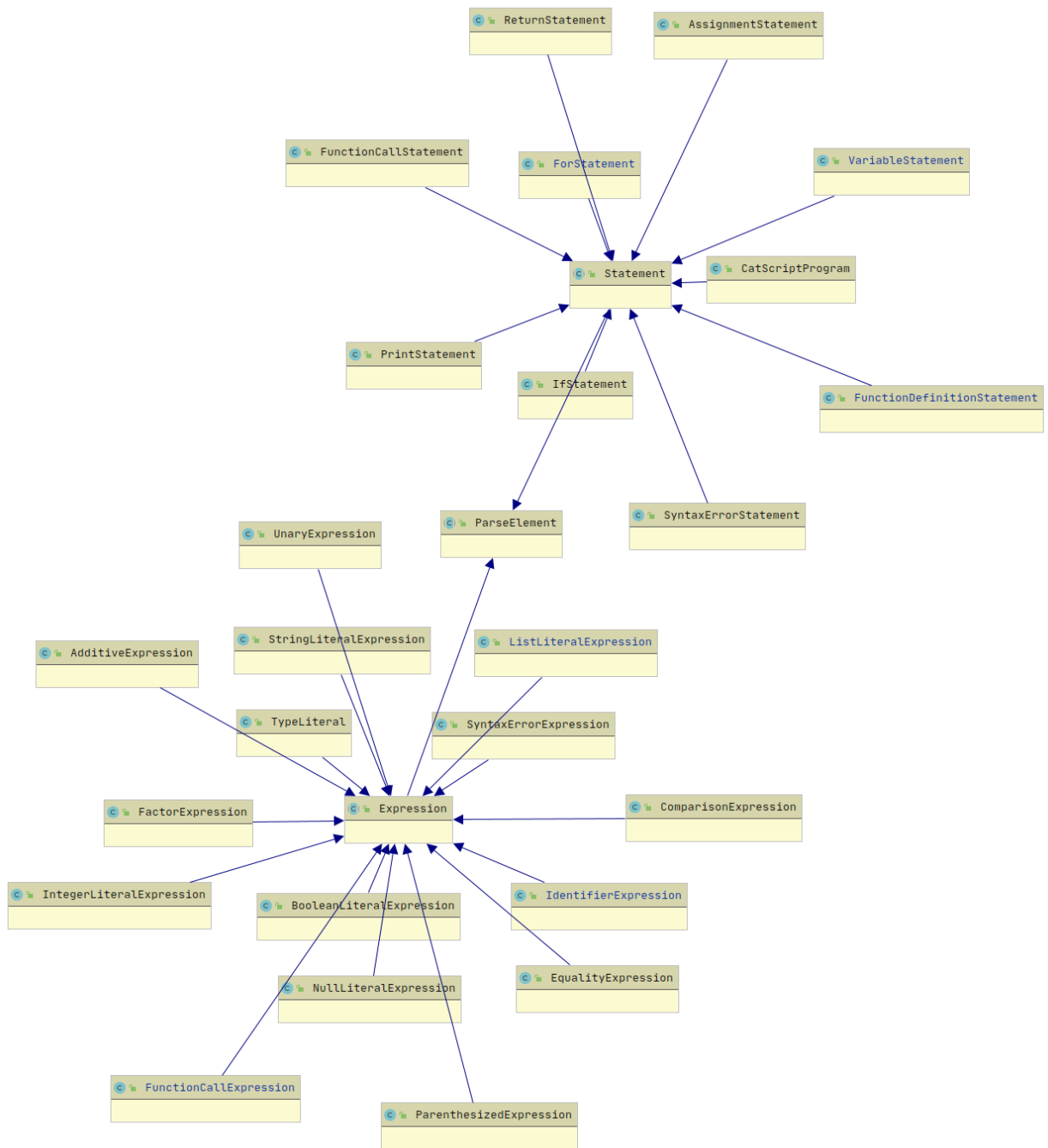
```
var implicit = 10
var explicit : string = "string"
```

### AssignmentStatement

The AssignmentStatement allows a value to be assigned to a variable. It has the following syntax: '*Identifier* = *Expression*,' where *Identifier* is an IdentifierExpression mapping to some variable, and *Expression* is some Expression, the value of which we will assign to this variable.

```
var assignable : string = "string"
assignable = "newstring"
```

## Section 5: UML



Powered by yFiles

This UML diagram represents the `ParseElement` class and its descendants. These are all the program elements that the Catscript parser can parse. There are two main parent classes that inherit from the `ParseElement` class. These are `Expression` and `Statement`. These are the two main categories of program elements that one can have in a Catscript program. The main difference between an expression and a statement is that an expression is something that evaluates to a value and a statement is something that executes. The classes that inherit from the



Expression class are all the different expressions that can exist in a Catscript program and likewise for the Statement class.

## **Section 6: Design Trade-Off**

For this project I made the design trade-off to create a recursive descent parser rather than a parser generator for Catscript. This decision was made because the process of creating a recursive descent parser makes the recursive nature of programming language grammars apparent and understandable. Had we used a parser generator instead, the process of creating the parse tree would have been obfuscated and much less intuitive than in the case of a recursive descent parser where the process is tangible. Despite it being more lines of code to create a compiler using a recursive descent parser, the benefits of gaining a window into the functionality of a programming language make it the ideal parsing method for this project.

## **Section 7: Life Cycle Model**

For this project I followed a Test Driven Development (TDD) development life cycle model. In this model, I began with a test suite of failing tests which should all pass when the project is complete and worked to make each test pass. This lifecycle made my progress very clear to me. At any time I could compute my completion based on the ratio of passing tests to total tests. This was very beneficial for monitoring my progress. Additionally, the tests aided my development by atomizing the tasks to be completed. It was much easier for me to sit down to complete a finite number of tests than it would have been for me to sit down to write an entire recursive descent parser. By doing this, the TDD lifecycle made this large project much less intimidating.