

# *CAPSTONE PORTFOLIO*

*Abigail Hustis and Bryndon Wilkerson*

*CSCI 468 | Compilers*

## Section 1: Program

[source.zip](#)

## Section 2: Teamwork

For this project my team consisted of me and one other person, Bryndon Wilkerson – team member 2. I was the primary coder of this project and contributed about 95% of the work that went into completing this project. Team member 2 was the primary tester of this project and supplied multiple tests for us to run to validate the quality and the efficiency of the codebase. Team member 2 also provided me with the documentation – located in Section 4 – of the Catscript language that helped with the writing of the codebase.

## Section 3: Design Pattern

One of the design patterns that we used while completing this project was the Flyweight design pattern which can also be called memoization. One of the main goals of this pattern is to optimize the amount of memory that we are using as well as the run time. The way that this is implemented is to store data in some data structure in a way that optimizes that use of shared data between two objects. One example in the Catscript Codebase would be to have a HashMap that acts like a caching system. This allows for the same information to be accessed by different objects at different times without taking up more space than you need. Here you can see in Figure 1 that each time that the `getListType()` method is getting called, we are creating an entirely new instance of the `ListType()` class.

```
public static CatscriptType getListType(CatscriptType type) {  
    return new ListType(type);  
}
```

Figure 2: Catscript method before memoization

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();  
public static CatscriptType getListType(CatscriptType type) {  
    ListType listType = cache.get(type);  
    if(listType == null){  
        listType = new ListType(type);  
        cache.put(type, listType);  
    }  
    return listType;  
}
```

Figure 1: Catscript method after memoization

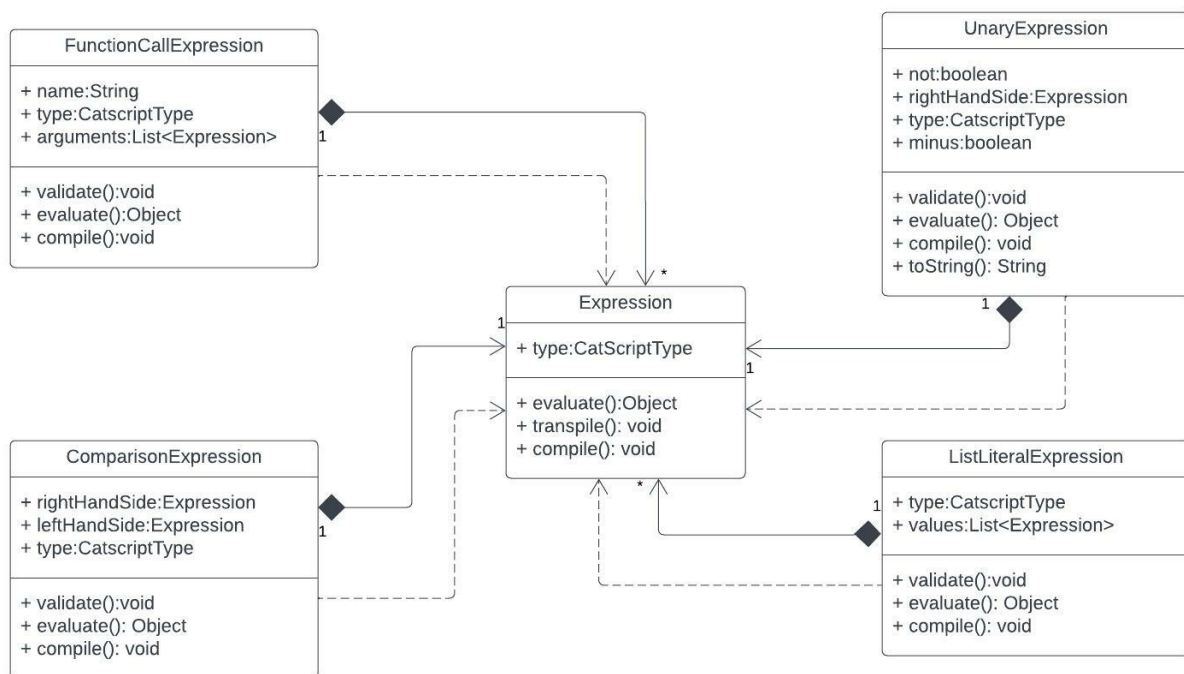
In this example, we are looking at how to access the type data of a list literal that we are creating. Instead of instantiating an entirely new Class object, we decide to memoize this code by

creating a HashMap where we can store the relationship between the CatscriptType and its correlated ListType. This way if we have already used the CatscriptType before we can simply access that HashMap and we are able to save time and space. Subsequently, if the CatscriptType has yet to be mapped to a ListType, we insert it into the HashMap, so we are increasing its usability.

## Section 4: Technical Writing

The documentation that was provided by team member 2 is located at the end of the portfolio.

## Section 5: UML



Here is a small excerpt of what a Class Diagram for this Codebase would look like. Expression is used as the base class for all the different types of Expression that Catscript supports. All of the expression classes depend on the super class Expression for any method that is not explicitly written. They all also have different relationships when it comes to the composition relationship that is present between all the Expression classes and Expression. For instance, ListLiteral and FunctionCall have a one-to-many relationship because they have the capability to implement a List of Expressions. While Comparison and Unary both have a one-to-one relationship because they will only represent a single instance of an Expression.

## Section 6: Design Trade-offs

We choose to write our compiler using recursive descent over using a parser generator. Recursive descent is top-down recursion where you start by parsing the highest-level expression or statement first and moving down the grammar as needed. As you move through the parser, you will filter down to the statement/expression that you need using conditional and, of course, recursion. One of the reasons that we choose this was because it gave a clearer picture of the recursive nature of the grammar. We were able to gain a greater understanding of the language that we were writing and were able to see the simplicity that recursion brings to a parser. Another reason that we made the decision to implement recursive descent is that it gives you a lot more control over the parsing process by allowing you to decide what the conditions are for each statement and when the parser goes to one type of statement or the other. It also allows you to have a more specific set of errors and total control over when and what errors are thrown.

## Section 7: Software Development Life Cycle Model

The software development life cycle model that we implemented for this project was test driven development. Throughout the course of our development, we solely relied on the test that were provided to us to ensure that our code was meeting all the requirements that we were given. The idea behind this style of development life cycle is to get rid of any ambiguity that your code is doing what you think that it is doing. It can be time consuming to try and run your code and to test all the different inputs and scenarios that might occur to insure robustness within the program. Tests provide that clarity and are quick to show you what use cases are not being met and what parts of your program is failing. In addition, tests can provide a sense of reassurance – once your test is passing, you can move on – and you don't need to worry about missing anything.

I enjoyed the test-driven model tremendously. Especially, jumping into a codebase that was already built out a bit, it was overwhelming at first to know where the best place was to start. The tests provided a great jumping off point because it gave us something that we could immediately debug and look at what the program was expecting the flow of the code to be. We were then able to start implementing features relatively quickly after starting to look at the codebase. For this project in particular the tests were extremely helpful in the sense that they required your code to be robust and optimized. As we progressed through the semester, the tests started to build off of one another so you had to make sure that they weren't only passing but you had written your code in a way that wouldn't break when you tried to implement the next step of the process.

# Catscript Guide

---

## Introduction

---

Catscript is a simple programming language containing various statements, function declarations, and expressions. Catscript compiles down to bytecode, which is then run on the JVM.

## Features

---

### Statements

A statement is a for loop, if statement, print statement, variable statement, assignment statement, or a function call.

```
statement = for_statement |  
if_statement |  
print_statement |  
variable_statement |  
assignment_statement |  
function_call_statement;
```

### For loops

A for loop iterates through all the values in a certain expression and executes a statement for each value

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',  
'{', { statement }, '}';
```

Example:

```
for(x in [1,2,3]) { print(x) };
```

### If statements

An if statement evaluates an expression. If the expression is true, then it executes a statement. If the expression is false and there is an else, it executes the body of the else. If there isn't an else, it moves on.

```
if_statement = 'if', '(', expression, ')', '{',  
{ statement },  
'}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

Example:

```
if (x == 2) { print(x) }  
else { print('x doesn't equal 2'); };
```

### Print statements

A print statement prints the value of an expression to the console.

```
print_statement = 'print', '(', expression, ')'
```

Example:

```
print(3);
```

### Variable statements

A variable statement declares an identifier and assigns an expression to the identifier. It may explicitly declare a type. If not, a type is implicitly decided.

```
variable_statement = 'var', IDENTIFIER,  
[':', type_expression, ] '=', expression;
```

Example:

```
var x = 3;

var x : int = 3;
```

## Assignment statements

An assignment statement assigns an expression to an already declared identifier.

```
assignment_statement = IDENTIFIER, '=', expression;
```

Example:

```
x = 3;
```

## Function calls

A function call executes the statements contained in the function called.

```
function_call = IDENTIFIER, '(', argument_list , ')'
```

Example:

```
foo(1)
```

## Function declaration

A function declaration declares a new function with an identifier. It contains a function body statement. It may contain a parameter list and/or contain a type expression.

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +  
[ ':' + type_expression ], '{', { function_body_statement }, '');
```

Example:

```
function foo(x : int, y : String) { print(x) }
```

## Function body statements

A function body statements contains a list of statements that will execute when the function is called. It may contain a return statement.

```
function_body_statement = statement |  
return_statement;
```

Example:

```
var y = 2;

y = y + 2;

return y;
```

## Parameter lists

A parameter list is a list that contains one or more parameters.

```
parameter_list = [ parameter, {',' parameter } ];
```

Example:

```
[1,2,3]
```

## Parameters

A parameter is an identifier. A type may explicitly be declared.

```
parameter = IDENTIFIER [ , ':', type_expression ];
```

Example:

```
var m = 2;  
var foo : bool = false;
```

## Return statements

A return statement returns from a function. It may return an expression.

```
return_statement = 'return' [ , expression];
```

Example:

```
return 3;
```

## Expressions

An expression is an equality expression.

```
expression = equality_expression;
```

## Equality expressions

An equality expression contains at least one comparison expression. For every equality (==) or inequality (!=), then it will have another comparison expression.

```
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };
```

Example:

```
x == 4;  
  
y != true;
```

## Comparison expressions

A comparison expression contains at least one additive expression. For every greater than (>), less than (<), greater than or equal (<=), or less than or equal (>=), then it will have another additive expression.

```
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };
```

Example:

```
x > 3;  
  
w <= 5;  
  
foo < 4;
```

## Additive expressions

An additive expression contains at least one factor expression. For every addition (+) or subtraction (-), then it will have another factor expression.

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression };
```

Example:

```
x + 2;
```

```
3 - 7;
```

## Factor expressions

A factor expression contains at least one unary expression. For every division (/) or multiplication (\*), it will have another unary expression.

```
factor_expression = unary_expression { ("/" | "*" ) unary_expression };
```

Example:

```
11 / 5;
```

```
12 * 2;
```

## Unary expressions

A unary expression either contains a "not" or negative (-) and a unary expression, or it contains a primary expression.

```
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
```

Example:

```
not true;
```

```
-4;
```

## Primary expressions

A primary expression contains an identifier, string, integer, true, false, null, list literal, function call, or an expression. \* int - a 32 bit integer \* string - a java-style string \* bool - a boolean value \* list - a list of value with the type 'x' \* null - the null type \* object - any type of value

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |  
list_literal | function_call | "(", expression, ")"
```

Example:

```
2
```

```
false
```

```
'foo'
```

```
null
```

```
[1,2,3]
```

## List literals

A list literal is a list containing one or more expressions.

```
list_literal = '[', expression, { ',', expression } ']';
```

Example:

```
[1,2,3];
```

## Argument lists

An argument list contains a list of one or more expressions.



```
argument_list = [ expression , { ',' , expression } ]
```

Example:

```
foo(1,2,3)
```

## Type expressions

A type expression is an int, string, bool, object, or list. If it is a list, it may contain a type expression.

```
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression, '>']
```

Example:

```
int
```

```
string
```

```
list<object>
```