

Montana State University  
CSCI 468 Compilers Capstone  
Member 1: Ali Khaef  
Member 2: Joe Burke

## Section 1: Program

The source code can be found in capstone/portfolio/source.zip

## Section 2: Teamwork

Team member 1: Ali khaef, did the majority of the coding which consists of tokenizer, parser, evaluation, and bytecode generation. The time spent for team member 1 was 95%.

Team member 2: Joe Burke, did the documentation regarding section 4 and wrote 3 extra tests which are to be found inside the demo folder and it is called "BurkeTests". The time spent for team member 2 was 5%.

## Section 3: Design Pattern

For this project I used memoization pattern which is in root/src/main/java/edu.montana.csci.csci468/parser/CatscriptType. which is in the picture below.

In the memoization pattern we store the catscript types in memory so that we don't have to calculate the listType again. Memoization is an optimised technique which will help us to make our program do different things faster.

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

## Section 4: Technical writing

# CatScript Documentation

---

## Introduction

CatScript is a simple, statically typed toy scripting language. It was created for educational purposes, but it can both evaluate and compile a pretty extensive amount of things (although lacking many of the 'core' features of more robust languages).

Ex:

```
var x = "foo"  
print(x) // prints "foo"
```

## Types

As mentioned above, CatScript is statically typed. The type system includes the following:

- **int**: A 32-bit signed integer.
- **string**: A Java-style string.
- **bool**: A Boolean value.
- **list**: An immutable list of values of any object type or subtype.
- **null**: The null type.
- **object**: A parent type of any of the above types.

## Commenting

CatScript supports both single-line and multi-line comments. Single-line comments begin with a `//` and continue to the end of the line. Multi-line comments begin with a `/*` and end with a `*/`.

Ex:

```
// This is a single-line comment.  
  
/*  
    This is a multi-line comment.  
    It can span multiple lines.  
*/
```

# Expressions

## Additive

CatScript supports an overloaded `+` operator for evaluating two objects. In CatScript you can:

- Take two ints and evaluate to their sum of type int.
- Take a string and another int/string/null object and evaluate to their concatenation of type string.

Ex:

```
1 + 1           // evaluates to 2
1 + "foo"       // evaluates to "1foo"
"foo" + "bar"   // evaluates to "foobar"
"foo" + null    // evaluates to "foonull"
```

## Comparison

There are four comparison operators:

`<` `<=` `>` `>=`

Only ints can be compared, and they evaluate to a bool type.

Ex:

```
1 < 2           // evaluates to true
1 <= 1          // evaluates to true
1 > 1           // evaluates to false
1 >= 1          // evaluates to true
```

## Equality

Compares two objects using one of two equality operators and evaluates to a bool type.

Equality operators are as follows:

`==` evaluates to true if the two objects are equal.

`!=` evaluates to true if the two objects are not equal.

Ex:

```
1 == 1           // evaluates to true
1 == "1"         // evaluates to false
"foo" == "foo"   // evaluates to true
"foo" == "bar"   // evaluates to false
1 != 2           // evaluates to true
```

## Factor

Evaluates to the result of multiplying or dividing an int by another int. As CatScript only support ints, the result of division will always be the truncated int.

Factor operators are as follows:

*\** evaluates to the result of multiplying the two ints.

*/* evaluates to the result of dividing the two ints.

Ex:

```
1 * 2           // evaluates to 2
2 / 1           // evaluates to 2
```

## Function Call

Evaluates to the result of the execution of a defined function with the given (including zero) arguments.

Ex:

```
function foo(x : int, y : int) {
    return x + y
}

foo(1, 2) // the function call -- evaluates to 3
```

## Identifier

Identifiers are used to reference a variable or function. They evaluate to the value of the variable or the execution of the function they are referencing.

Ex:

```
var x = 1 // x is the identifier here and evaluates to 1
```

## List

A list is an immutable collection of any object type or subtype.

The element types of a list are optionally specified by the `<` and `>` symbols for type guarantees.

Ex:

```
var x = [1, 2, 3] // x is a list of ints

function foo(l : list<int>) { // foo is a function that takes a list of ints.
  Notice the type specification.
  print(l) // prints [1, 2, 3]
}
```

If no type is specified, the list type will be inferred. If multiple types are found, the type of the list will be of type `object`. Note that passing a list into a function without specifying type in the parameter of the function, the function will assume the type of the list is `object`.

## Unary

Evaluates to the opposite of an int literal or a boolean literal. Uses the unary operators:

- negates an int literal.

`not` negates a boolean literal.

```
-1 // evaluates to -1
-(2 + 3) // evaluates to -5
not true // evaluates to false
```

## Parentheses

All expressions evaluate to the result of the operation they are in, and follow an order of operations. Parentheses are used to force evaluation of an (perhaps nested) expression before it is used in its parent expression (or final evaluation, if no parent exists).

Ex:

```
1 + 1 * 2    // evaluates to 3
(1 + 1) * 2  // evaluates to 4
```

## Literals

Here are the literal types in CatScript. These are just the types that are supported by the language, and the way they need to be written in code in order for the parser to recognize them.

- int literal - simply a number without a decimal.
- string literal - a string of characters surrounded by double quotes.
- bool literal - either **true** or **false**, must be lowercase.
- null literal - just the word **null**, must be lowercase.
- list literal - a list of objects surrounded by square brackets and separated by commas.

The way to represent these types are as follows:

```
1          // an int literal, simply a number
"foo"      // a string literal, a string of characters with double quotes
around it
true       // a bool literal, either true or false -- lowercase
null       // a null literal, lowercase
[1, 2, 3]   // a list literal of ints
["foo", "bar"] // a list literal of strings
[1, true, "1"] // a list literal of objects
```

## Statements

### Assignment

Sets, or re-sets, the value of a variable by copying the value of an expression into the variable itself. It is invoked by the assignment operator **=** between an identifier (variable) and an expression. Note that while similar to a

variable declaration, an assignment statement is not a declaration (although they can often happen together). The variable must be declared before it can be assigned.

Ex:

```
x = 2           // this is setting, or re-setting the value of x to be 2
x = x + 1       // this is re-setting the value of x to be the value of the
                // expression x + 1, or 3

print(x + 10)   // prints 13
```

## For Loop

The for loop is used to iterate over every element in a list. This loop is invoked using the **for** keyword, followed by a variable name, followed by the **in** keyword, followed by the list name (or a list literal itself).

Ex:

```
// using a variable for the list
var x = [1, 2, 3]
for (i in x) {
    print(i) // prints 1, 2, 3
}

// using a list literal
for (i in [1, 2, 3]) {
    print(i) // prints 1, 2, 3
}
```

## Function Definition

The function definition statement is used to save a function to an identifier so that it can be called at any point in the future within that scope.

It is invoked by the **function** keyword, followed by the name of the function, followed by the parameters of the function, each parameter separated by commas, and all enclosed in parentheses, followed by the body of the function enclosed in curly braces.

- The parameters of the function are the names of the variables that are passed into the function.
- The body of the function is the code that is executed when the function is called.



Ex:

```
function foo(x : int, y : int) {  
    return x + y  
}
```

The name of the function here is **foo**.

The parameters of the function are **x** and **y**, optionally specifying the type of the parameter as **ints**.

The body of the function is **return x + y**, which returns the sum of the two parameters.

## If Statement

The if statement is used to execute a block of code based on the value of a boolean expression being true or false. It is invoked by the **if** keyword, followed by the boolean expression to be checked, followed by the body of the if statement enclosed in curly braces.

An if statement optionally has an else statement, which is invoked by the **else** keyword following the last curly brace of the **if** statement, followed by the body of the else statement enclosed in curly braces.

It will look something like this: **if (x > y) { ... } else { ... }**. This is saying that if the expression **x > y** is true, then execute the code in the curly braces, otherwise execute the code in the curly braces following the **else** keyword.

Ex:

```
if (x > y) {  
    print(x)  
} else {  
    print(y)  
}
```

## Print

Takes the value of an expression and places it on the console, followed by a newline character. It is invoked by the **print** keyword, followed by the expression to be printed.

Ex:

```
print(1 + 2)
```

This will put the value of the expression `1 + 2` on the console. In other words, it will print `3` followed by a newline character to the console.

## Return

The return statement is used to return the value of an expression from a function. It is invoked by the `return` keyword, followed by the expression to be returned.

"Returning" simply means that the function will stop executing and "give back" the value of the expression.

The return statement is optional, and if it is not present, the function will return `null`.

Ex:

```
function foo(x : int) {  
    return x + 1  
  
    print("bar") // this will never be printed  
}  
  
var number = foo(2)  
  
print(number)
```

So the function will return the value of `x + 1`, which in this case is `3` since we've called `foo` with a value of `2`. The function will not print `bar` to the console because it appears after the return statement and since return stops execution when it is called, the print statement in the function will never be called.

## Variable

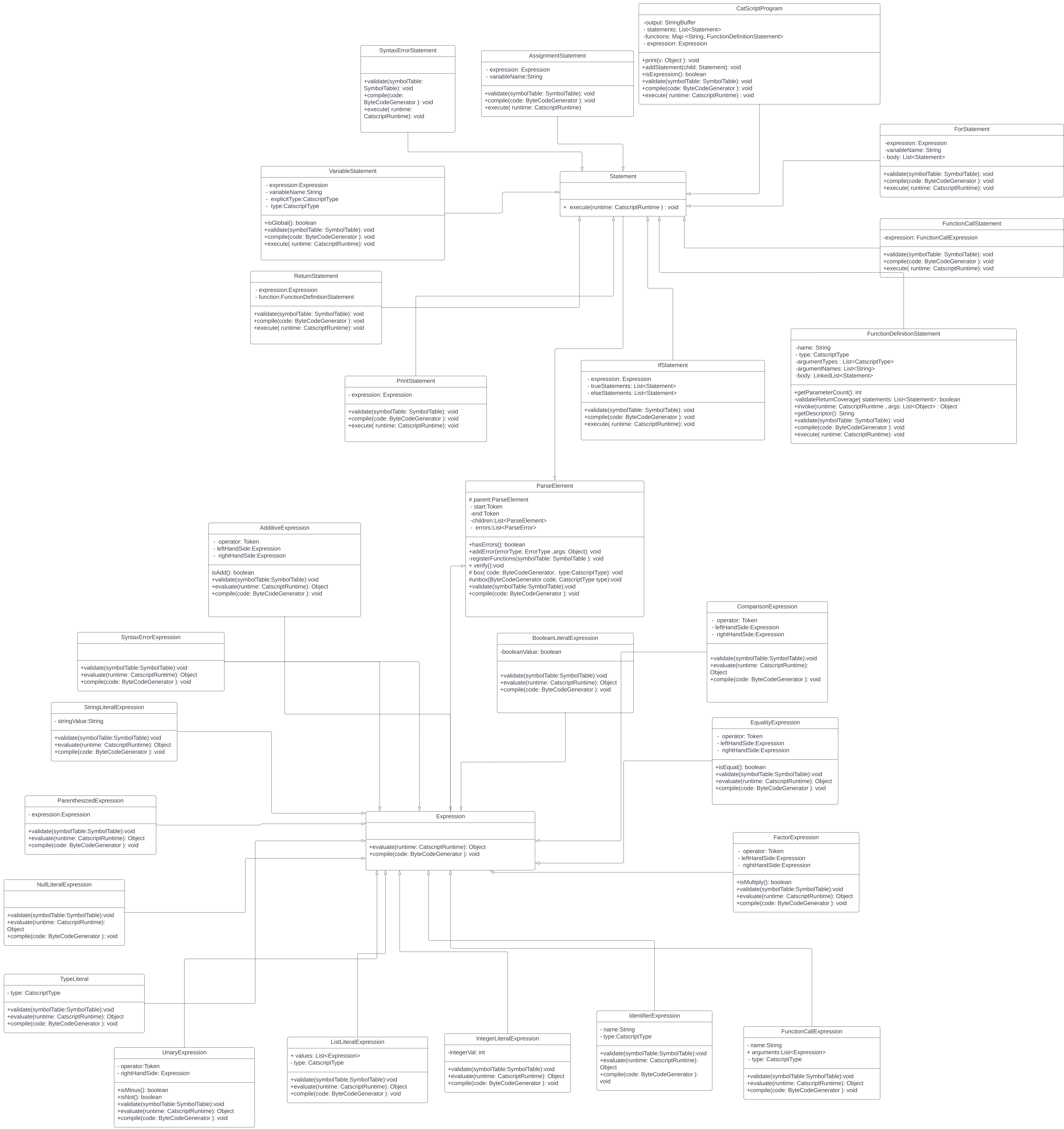
The variable statement is used to declare a variable. It is invoked by the `var` keyword, followed by the name of the variable, followed by an `=` operator, followed by the expression to be assigned to the variable. This is similar to an assignment statement, except that the variable is being declared or defined before the assignment.

Ex:

```
var x = 2           // this is a variable declaration, or definition, followed by  
an assignment  
x = 1              // this is an assignment statement
```

## **Section 5: UML**

The two main and important classes in this UML Diagram are Statement and Expression classes. All the other classes except ParseElement will inherit from one of these classes. On the other hand, Statement and Expression classes will inherit from ParseElement.



## **Section 6: Design trade-offs**

Catscript is a statically typed language, meaning that the types of variables are determined at compile time. variables cannot be reassigned to different types. if objects are implemented in the future, users must define classes similar to java. variable types do not need to be explicitly written which can be interpreted as trade off because users might think variables can be dynamic. statically typed languages have a slower compile time but faster run time than interpretive languages. Another topic that is important to be talked about in this project is choosing between recursive descent and parser generator. In this project we used the recursive descent parsing. In general the parser generator (EBNF) is going to be faster but using a recursive descent parser will give us a better control on our code and it is going to be easier to debug the code.

## **Section 7: Software development life cycle model**

We used Test Driven Development in this project. We knew the features and requirements ahead of time and simply had to implement them. because the desired outcome was clear, implementation was straight-forward. Since the tests were written by developers, the feature set was limited. The test just tested the output and not that tokens were correctly distinguished and parsed. the next thing i would implement if i were to deploy this would be dictionaries as a native catscript literal.