

Montana State University

Gianforte School of Computing

Computer Science - Professional

CSCI 468 Capstone Report

**Team Members**

Colton Weeding  
Logan Dolechek

**Project Advisor**

Carson Gross

**Semester**

Spring 2022

# Section 1: Program

---

The source file for this program is located in the same directory as this document named `source.zip`.

## Section 2: Teamwork

---

Each team member was responsible for writing their own code throughout the course of the semester. Writing the code for the compiler took on average 10 hours per week for 13 weeks. This comes out to 130 hours for the compiler alone.

Each team member was tasked with contributing 2 technical documents, the Catscript Documentation and unit tests, to the other team members portfolio. Each team member spent between 10 - 15 hours on the creation of the technical documents.

In total 145 hours were dedicated over the course of 1 semester to this Capstone project. The accompanying documents were written by Logan Dolechek.

This test was written to ensure the functionality of parenthesis when using an equality expression.

```
@Test
void parenthesizedEqualityExpressionEvaluatesProperly() {
    assertEquals( expected: true, evaluateExpression( src: "(true) == true" ));
    assertEquals( expected: true, evaluateExpression( src: "4 + (2 * 3) == 10" ));
    assertEquals( expected: true, evaluateExpression( src: "not true == false" ));
}
```

This test was written to ensure the robustness of the for loops by making sure they can handle lists with different types.

```
@Test
void forStatementWithDiffTypesWorksProperly() {
    assertEquals( expected: "8\nasdf\n5\ntrue\nnull\n", executeProgram( src: "for(x in [8, \"asdf\", (3+2), true, null]) { print(x) }" ));
}
```

This test was written to test the functionality of return statements with parenthesized expressions.

```
@Test
void returnListWithListWorksProperly() {
    assertEquals( expected: "[6, 10, 20]\n", executeProgram( src: "function foo() : List<int> { return [(2*3), (12-2), (18+2)] }" + "print(foo())" ));
}
```

## Section 3: Design Pattern

---

One of the design patterns we used in our Catscript compiler was the memoization or flywheel pattern. Memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

This pattern was used when calling to the Catscript type system when getting a list type. A cache was created using a HashMap that could hold the types of lists. The call started by checking the type of the list and only adding a new type to the cache, if it was not there already. This was done to reduce the amount of calls to the Catscript type system. While this was effective in Catscripts' case, it is not thread safe code. If Catscript were to be a multi-threaded application we could have the chance of running into logical errors. To make this thread safe code we chose to use a ConcurrentHashMap and the function `.computeIfAbsent` to avoid logic errors.

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

# Section 4: Technical Writing

---

## Introduction

This is the documentation for the Catscript scripting language. The following features are categorized by expressions and statements. Statements are executed, while expressions are not. For this reason the print statement is used to demonstrate many of the expressions. Expressions also happen to be the building blocks of many of the statements, so that is why expressions are defined first.

CatScript is statically typed, with a small type system as follows

- 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

Each main expression/statement will have a code example with the following format

- Lines starting with '>>' will indicate the code being executed
- Lines without '>>' will be the resulting output of the above lines

## String Literal Expression

Catscript supports string similarly to other languages

- to make a string use the double quotes ("" )

```
>> "hello world"
```

This will evaluate to “hello world”

## Integer Literal Expression

Catscript only supports integers and not floating points

```
>> 1
```

This will not be evaluated to 1

```
>> 5.8
```

This will not be evaluated and will result in an error

## Boolean Literal Expression

The supported Boolean expression of Catscript are

- true
- false

Boolean values can be set to variables and used for comparisons

```
>> true
```

This will be evaluated as true

```
>> false
```

This will be evaluated as false

## Null Literal Expression

```
Null
```

This will evaluate to null

## Additive and Factor Expressions

The additive and factor expressions supported by Catscript are

- addition (+)
- subtraction (-)
- multiplication (\*)
- division (/)

Subtraction, multiplication, and division are only supported with integers.

```
>> 2 + 2
```

This will evaluate to 4

```
>> 5 - 4
```

This will evaluate to 1

```
>> 10 / 2
```

This will evaluate to 5

```
>> 4 * 5
```

This will evaluate to 20

Additionally the addition (+) operator can be used for string concatenation

```
>> "hello" + " world"
```

This will evaluate to "hello world"

```
>> 2 + " string"
```

This will evaluate to 2string

## Unary Expression

The unary expressions supported by Catscript are

- negative (-)
- negation (not)

```
>> -1
```

This will evaluate to -1

```
>> not true
```

This will evaluate to false

```
>> not not true
```

This will evaluate to true

## Equality Expression

Equality expressions allow you to check if two integer values are the same and return the Boolean value True if the expression is true and False if the expression is false.

```
>> 2 == 2
```

This will evaluate to true

```
>> 2 != 3
```

This will evaluate to true



## Comparison Expression

Comparison Expressions allow you to compare two integer values and will return the Boolean value True if the expression is true and False if the expression is false. The comparisons that are supported are

- less than (<)
- greater than (>)
- less than or equal to (<=)
- greater than or equal to (>=)

```
>> 2 > 1
```

This will evaluate true

```
>> 1 < 2
```

This will evaluate to true

```
>> 2 < 1
```

This will evaluate to true

```
>> 2 >= 1
```

This will evaluate to true

```
>> 1 >= 2
```

This will evaluate to false

## List Literal Expression

List Literal Expressions in Catscript allow you to group objects together. The nice thing about Cats Scripts lists is that they are allowed to contain object with different typing, so the whole list does not have to stay to one type.

```
>> ["a", "b", "c"]
```

This will evaluate to ["a", "b", "c"]

```
>> [5, 6, 7]
```

This will evaluate to [5, 6, 7]

You can also include different types inside of a list and even put lists inside of a list

```
>> [true, false, 1, 2, "here", [8, 9]]
```

This will be evaluate to [true, false, 1, 2, "here, [8,9]]

## Parenthesized Expression

Catscript allows you to parenthesize mathematical operations to indicate precedence. By default mathematical operations are left associative.

```
>> 2 + 3 * 2
```

This will evaluate to 10

```
>> 2 + (3 * 2)
```

This will evaluate to 8

## Function Call Expression

```
x = func1(1, 2, 3)
```

This will call the function 'func1' with 3 parameters being passed into it. The value of x will be the result of the return statement inside of 'func1'

```
func1()
```

Not all functions need parameters, so not all function call expressions need to pass in parameters. Similarly, not all functions return a value. In this case the function 'func1' will be called with no parameters and no return value being stored in a variable

# Statements

---

## Print Statement

The print statement allows you to

The format for the print statement is as follows

- keyword 'print'
- item to be printed enclosed in parenthesis '()'

```
>> print(1)
1
>> print("hello world")
hello world
```

## Variable Statement

The variable statement can be used to store a value and give it a name inside of Catscript. You can either explicitly declare the typing of the value, or Catscript will automatically determine the type. The supported types are

- Integer (int)
- Boolean (bool)
- String (string)
- Object (object)
- List (list<>)

The format for a variable statement is as follows

- keyword (var)
- the name of your variable
- *optional* semicolon (:) followed by a type (int, bool, string)
- equals sign (=)
- the value being assigned to the variable

```
>> var b = 15
>> print(b)
15
```

```
>> var x : int = 10
>> print(x)
10
>> var y : bool = true
>> print(y)
true
>> var z : string = "hello world"
>> print(z)
hello world
```

Note that if you explicitly declare a type and the value you are trying to assign does not match the type, you will receive an error.

## Assignment Statement

The assignment statement allows you to reassign a variable that has already been created to another value.

```
>> var x = 2
>> print(x)
2
>> var x = "hello world"
>> print(x)
Hello world
```

## For Statement

The For Statement in Catscript iterates through a set of values.

The format for a for statement is as follows.

- Keyword 'for' followed by an opening parenthesis '('
- Name of iterator, keyword 'in', list to iterate through, closing parenthesis ')'
- Opening curly brace '{'
- Operation to execute for each step in iteration
- Closing curly brace '}'

```
>> for (i in [1, "a", 3]) {
>>   print(i)
>> }
1
a
3
```

## If Statement

The if statement allows you to set an operation to be executed if a given comparison expression evaluates to true. The format of an if statement is as follows

- keyword 'if'
- comparison expression enclosed in parenthesis '()'
- operation to be executed enclosed in curly braces '{}'

Additionally an else statement can be added to any if statement as well. This will add another operation to be executed in the case of the original if the statement fails. The format for an else statement is as follows

- keyword 'else' after the closing curly brace '}' of the if statement
- operation to be executed enclosed in curly braces '{}'

```
>> if (11 > 10) {  
>>   print("11 is greater than 10")  
>> }else {  
>>   print("10 is greater than 11")  
>> }  
11 is greater than 10
```

## Function Call Statement

The function call statement allows you to create a function that can be called at a different point by using the name assigned to it and passing the correct variables to it.

The format for creating a function in Catscript is as follows

- keyword 'function'
- name of function
- **optional** semicolon ':' followed by a type to specify the return type of the function
- opening and closing parenthesis '()' containing the variables you expect to pass into the function.
  - NOTE: you can also specify the type of each of the variables by including a semicolon ':' after its name followed by the type
- opening curly brace '{' to start the body of the function

- fill the body of the function with the process that you wish to be executed upon calling the function
- an optional Return Statement can be added as the last line of the body if you wish for the function to return a value
- closing curly brace '}'

```
>>function x: int (x: int, y: string, z: int) {
>>   print(y)
>>   return x + z
>> }
>> x(1, "abc", 2)
abc
>> print(x(1, "abc", 2))
abc
3
```

The first time the function is called nothing is done with the return value, so just the value of 'y' is printed

The second time the function is called the value of 'y' is printed, as well as the return value 'x + z' of 3

## Return Statement

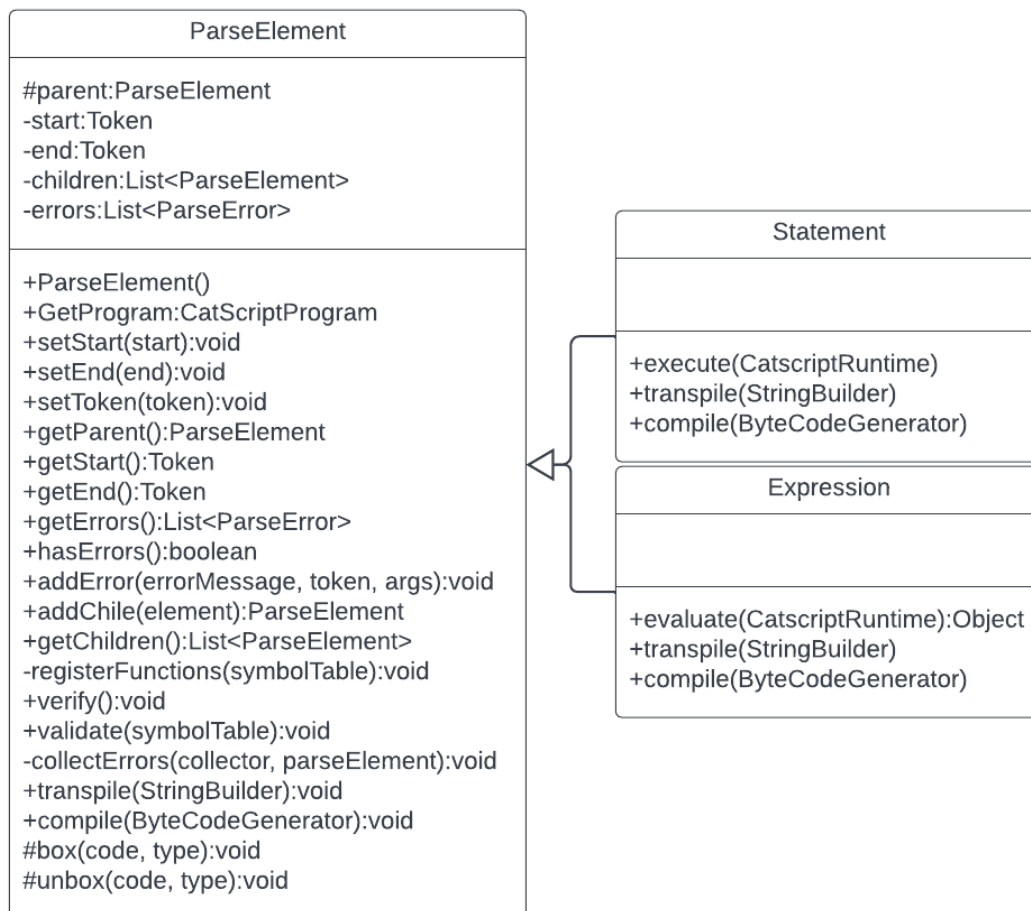
The return statement is used within a function body to specify a value or object that should be returned from the function

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

The function returns the value of 10, so the print statement prints that value.

# Section 5: UML

Included in this section is a model of the Catscript code starting from the 'Parse Element. The classes Statement and Expression both inherit from Parse Element. All of the Statements and Expressions discussed in the documentation inherit from the base classes of Statement and Expression.







# Section 6: Design Trade-Offs

---

## **Recursive Parsing vs. Parser Generator**

We chose to build a recursive descent parser instead of using a parser generator like previous classes. This allowed us to be able to better debug the compilers and see what was going on at each step. If we had chosen to use a parser generator there would have been a whole part of the compiler that we wouldn't have been able to make/see. By making a recursive descent parser it allowed us to understand the process of parsing a language and better understand where our errors were. While it would have been easier and faster to use a parser generator, I am glad that we chose to write the parser.

# Section 7: Software Development Life Cycle Model

---

## **Test Driven Development Cycle (TDD)**

With test driven development we had tests written that tested the functionality of different parts of the compiler and were separated into different steps. This helped by forcing us to complete the earlier parts correctly before moving onto the harder parts and thus minimized the amount of errors we had to go back and fix at the end.