

# **CSCI 468 Compilers**

## **Spring 2022 Capstone Report**

Jacob Sitch & Justin Scarbrough

## Section 1: Program

The source code for this project can be found in a zip file located in the capstone directory of the associated github repository at /capstone/portfolio/source.zip

## Section 2: Teamwork

### Teammate Participation:

My partner for this project is Jacob Sitch, who provided documentation for Catscript, as well as test cases for features that the course did not test itself. The tests provided will be shown in the following subheading, and documentation is shown in section 4. Unlike most capstone projects, the teamwork in this project was very limited to the point where we each worked on our own project almost completely individually, and then provided each other documentation and extra tests. Because of this, assigning a 'percentage of work' is difficult to calculate, and doesn't really show a good dynamic of work between us. For those reasons it will be omitted from this report.

### Testing:

The following tests were authored by and provided to me by my partner, and can be found in the source code in src/test/java/PartnerTest.java:

```
@Test
void ifElseTest() {
    //Effectively an if else statement
    assertEquals("1\n", executeProgram("" +
        "if(false) {\n" +
        "} else {\n" +
        "    if(true) {\n" +
        "        print(1)\n" +
        "    }\n" +
        "}\n"));
}

@Test
void NegativeNumbersTest() {
    //Tests that negative number computation works correctly
    assertEquals(-10, evaluateExpression("2 * -5"));
    assertEquals(10, evaluateExpression("-2 * -5"));
}

@Test
void ListTest() {
    //Tests that lists can hold multiple types
```

```

    assertEquals("[1, BooleanLiteralExpression,
StringLiteralExpression]\n",
executeProgram("print([1,true,\"hi\"])\n"));
}

```

These tests check three components of Catscript's functionality, and are explored in more detail below:

### *Else If Test*

The first test checks that if statements are properly set up to handle 'else if' conditions by executing an assertion to see if a printed value matches depending on the proper condition being met. In this case, the primary condition (false) is not met, but the second condition, else if (true) is met, so the test passes.

### *Negative Numbers Test*

The second test checks to see that negative numbers are properly multiplied by the factoring component of Catscript. This test has two different assertions, one where a positive integer (2) is multiplied by a negative integer (-5), resulting in a negative value (-10). The second assertion is two negative integers (-2 and -5) multiplied together, resulting in a positive value (10). Because the compiler is written to handle arithmetic in this way, the test passes.

### *List Typing Test*

The third test checks to see that lists are able to hold multiple types. This test consists of an assertion checking that the expected value of 1, as well as type values for a BooleanLiteralExpression and a StringLiteralExpression are displayed when printing a list consisting of the integer 1, the boolean value true, and the string "hi". This test was written incorrectly, as printing a list in catscript prints each element of the list, rather than its type. The expected value of '1' for the first element is correct, but rather than expecting types for the other two elements, the actual value of the element should be expected instead.

### *Fixing the Test*

One way to fix this test would be to write a new assertion expecting a true value and requiring an input of a list with different types for each element. Use a for loop to iterate through the list and check the type of the current element, storing the data of the previous type (not counting the first element of the list). After iterating through the list once, check the current type with the previous. If these types do not match, pass the test, as it shows that the list is capable of holding multiple types. If the for loop completes its iteration with all elements matching in type, fail the test. In this case, it's possible that the list input is all of one type, but to properly write this test, the list *must* be of different types, which means a failing test would only mean that the typing system for the compiler cannot identify different types, or that the list is not able to return the proper types of its contents. Alternatively, this test could be fixed by simply expecting the values [1,true,"hi"], as they are all of different types, though doing this might not definitively

prove that the compiler was able to determine that the types of these elements are indeed different.

#### Documentation:

My partner created documentation for Catscript, going over the language in a very basic and fundamental way. Each feature of the language was written out in explicit detail with examples of how to go about using them, or at least how they should appear syntactically while programming with Catscript. While a little lengthy, this documentation does a good job of explaining Catscript to someone who has had almost no previous experience with programming, which seems to fit well with the category of Catscript's intended use case, that being as a lightweight scripting language. This documentation is viewable in Section 4 of this report.

#### My Work:

My primary work on this project consisted of authoring code for tokenizing, parsing, and executing catscript code, as well as using provided tests to ensure that the program worked properly.

#### *Code Generation:*

Writing code for this project was a process that took place over the course of the entire semester. I was instructed on the material at hand, such as tokenizing and use of recursive descent parsing and was then given the task of programming the functionality for those same functions I was instructed on. The project was split into four checkpoints where certain parts of the code had to be completed by a certain date, and with each checkpoint I followed a similar pattern to write the code in a timely manner while also still being correct.

#### *Debugging and Testing:*

Done in tandem with writing the code, the use of Test Driven Design allowed me to complete this project in a way that consisted of writing samples of code, and then debugging them with the pre-written tests for each checkpoint. After these tests for one component of the code passed, I moved on to other components, using the information and patterns that I learned from the previous sections. For example, when writing code to get parsing functionality to work, I started by working on additive expressions (Such as  $1+1+1$ ). After I wrote the code to process additive expressions, I moved on to test factoring expressions (such as  $1*1*1$ ). Because the two are similar in nature, and I already wrote code that processes addition, altering it to handle multiplication and division was a very simple matter.

### **Section 3: Design pattern**

#### Design Pattern Used:

For this project we made use of the memoization pattern when accessing types for a list object. Without memoization, the method looked like this:

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

```
}
```

This implementation works fine, but is very inefficient, as it creates a new `ListType` object with every single call of the method, which can cause an unnecessary amount of memory allocation and potentially slow down a system. For example, if a list is made up of only `int` type data, but is 100 entries long, that's 99 entries of a list type that we don't really need to generate. This method is designed to return what type the data we're looking at is, so we don't need to generate new objects for each entry in a list, rather just for the first occurrence.

#### Memoization Explained:

Memoization is a design pattern that, put simply, uses caching to speed up computation. A table (usually a hashmap) is used to store new data entries, and is then called on when that same value appears. Memoization is typically used in sequences where data repeats often, which is what `getListType` does.

With memoization, our code looks like this:

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

This implementation uses a map to store `ListType` objects, and rather than creating a new object for any entry in a list, checks the map to see if one already exists. If it does exist, the method just returns that type (since they're the same). If it doesn't, the `ListType` is put into the map before being returned, so in the future no new object needs to be generated.

## **Section 4: Technical Writing**

The following documentation has been provided by my partner. This is written in a markdown format:

```
# Catscript Guide
```

```
This document should be used to create a guide for Catscript, to
satisfy capstone
```

requirement 4

## ## Introduction

Catscript is a simple scripting language. It tokenizes, parses, and evaluates input to then compile into code.

## ## Features

Catscript is built on a couple data types, ints, strings, lists, booleans, null, and objects. With these data types we are able to use Catscript to create Print statement, For statements, If statements, and Function statements.

## ## Expressions

### #### Literals

Catscript has multiple literal values. Integers, strings, lists\[x] of type x, boolean, null, and object values are all acceptable in Catscript. Catscript can also handle parenthesized expressions and negative numbers.

```
```\n1          //Integer\n\"Hello\"     //String\nlist[1,2]   //List[Integer]\ntrue       //Boolean\n```\n
```

### #### Additive/Factor

Catscript can add, subtract, multiply, and divide any integer given to it and correctly return the computed value.

```
```\n1 + 2 //Result of 3\n2 - 2 //Result of 0\n1 * 4 //Result of 4\n4 / 2 //Result of 2\n```\n
```

#### #### Comparison/Equality/Unary

Catscript has the ability to compare the value of integers to return a true or false value depending on the operator. It is also able to determine if an input is equivalent. Catscript can also take the unary value of 'not' to change a boolean value.

...

```
1 > 2    //true
4 >= 4   //true
3 <= 1   //false
2 < 4    //true
...
```

...

```
1 == 1           //true
false == false   //true
true != true     //false
...
```

...

```
not true         //false
not not true     //true
...
```

#### #### Function Call

A Function Call in Catscript allows you to call to a function somewhere in the code. Use the name of the function and the correct argument to call it.

...

```
func(x) //Result depends on functionality of function func()
...
```

#### ## Statements

#### #### Print Statement

Print statements in Castscript are similar to some popular languages in that it is called with print(value) and has the ability to concatenate strings and integers.

...

```

print(1)          //1
print(2 + 2)      //4
print("hello")    //hello
print("one" + 1)  //one1
` ``

```

#### #### Variable Statement

The Variable statement is able to set a value to a named variable on the symbolTable to use throughout the program.

```

` ``
var x = 1
print(x)          //1
` ``

```

#### #### For Statement

For statements in Catscript require a variable and a list in the form of for(variable in \[list]) { inner }. These can be used to iterate through a statement multiple times.

```

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

```

#### #### If Statement

If statements in Catscript require a true value to execute its inner code. An Else can be executed if the initial condition fails. It also accepts nested If statement.

```

` ``
if(true) { print(1) }           //1
if(1 + 1) { print(1) }         //1
if(false) { print(1) } else { print(2) } //2
` ``

```

#### #### Function Statement

Functions can be created in Catscript that can contain many different operations inside of it. Create a function with the keyword 'function' followed by the name of the function, in this case, 'func'. You can then set the arguments required



for your function, either implicitly or not. You can then code the inner part of your function.

```

...
function func(x : int) { print(x) } //prints input for x
function func2() { print("hello") } //prints 'hello'
...

```

### #### Return Statement

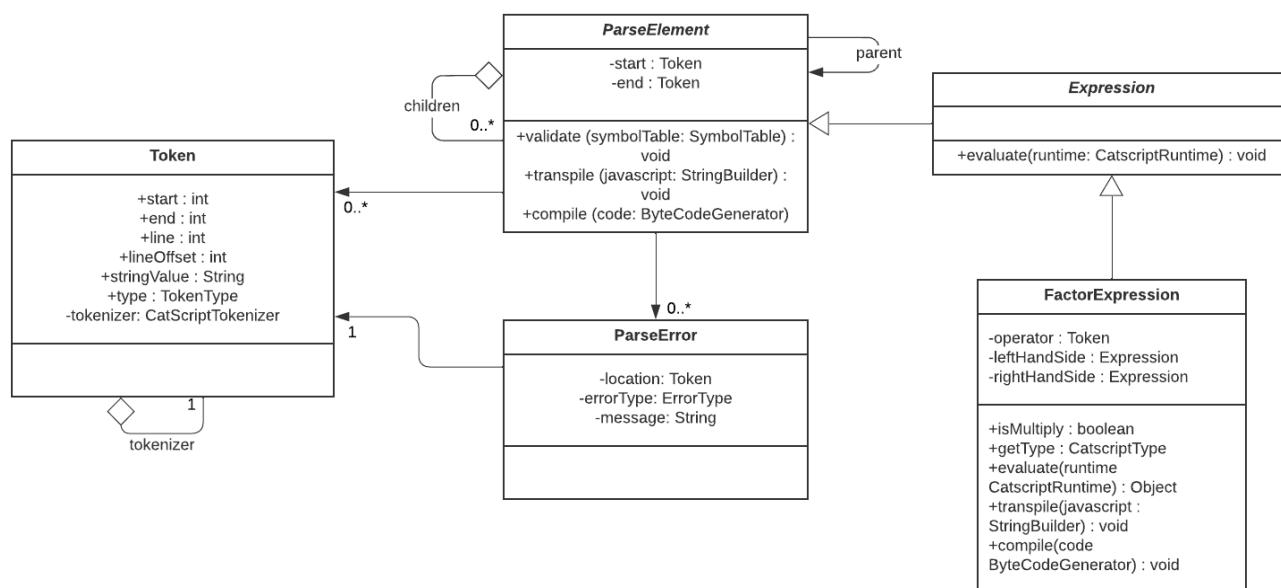
A function in catscript can also return a value to the function call. That value can then be used wherever it was called from.

```

...
function func3(x) { return x + 1 } //Returns value of argument plus
1
val 4plus1 = func3(4)                //4plus1 == 5
...

```

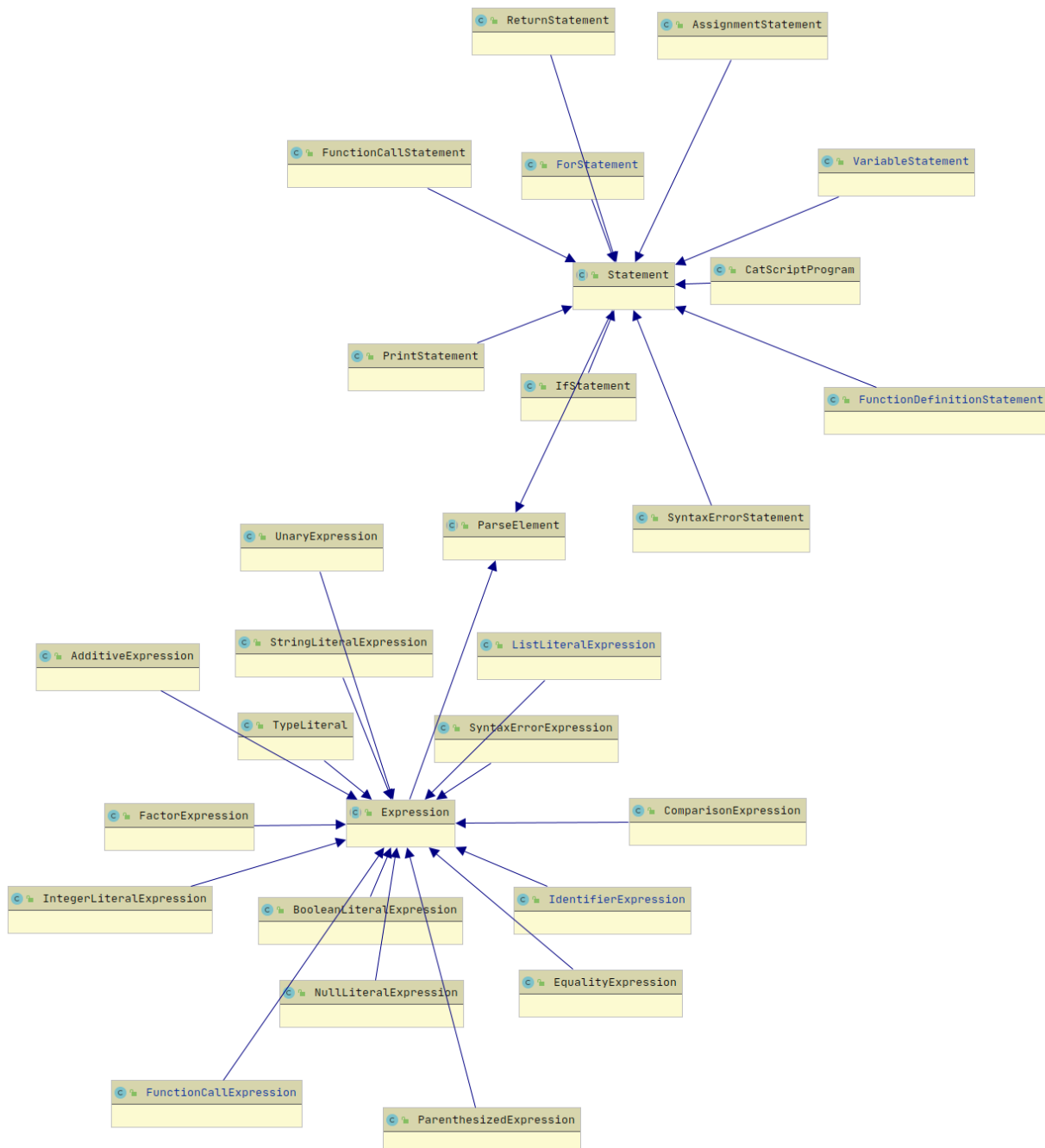
## Section 5: UML



The following is a class diagram for the FactorExpression class. This class diagram shows how the FactorExpression class relates to its interface, expression, which extends other classes. The classes that ParseElement has access to, ParseError and Token, should have accessor methods due to the private nature of their variables, but that goes without saying. One thing worth noting with the FactorExpression class diagram is that it will also potentially call itself, as well as any other expression. This was omitted from the class diagram due to the confusing

nature of recursion in UML. FactorExpression's extended classes ParseElement and Token also have access to instances of their own classes, though these are in limited quantities. In the case of Token, the tokenizer variable will only have one instance, and ParseElement's children could be any number, though it's not through recursion (directly) that these children are generated.

The following is a provided UML class diagram showing the compiler's overall structure and class relation:



This UML diagram is sparsely populated compared to the one for FactorExpression, though this is because the diagram is displaying relations between each class that has a connection to both Expression and Statement. Most other classes for expressions and statements likely follow the same pattern seen in FactorExpression. For the sake of brevity they will not be explicitly displayed.

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

### Recursive Descent and Parser Generators

This project saw a few design trade-offs, namely in the different ways to design a compiler. The two methods discussed in the course were parse generator compilers and recursive descent compilers. At a very broad level, recursive descent is a practice that uses recursion, as the name implies, to parse through lines of code from a certain starting point, usually an identifier or symbol relating to an expression. A parser generator, on the other hand, is created with a much more explicitly defined grammar in mind. Due to these stricter rules, parser generators are usually much more involved and detailed than a recursive descent compiler.

The main trade-off seen with using a recursive descent compiler is that of convenience and intuitive design at the cost of performance and efficiency. Recursive descent compilers typically are not as capable as parser generators due to the fact that there's no specific grammar being anticipated (beyond basic ideas). Because a general pattern is applied to parsing and tokenizing with recursive descent, grammar can be expected, but the means of compiling will probably take longer than using a parser generator specifically designed to handle the grammar needed to compile a certain language. In the scope of this class, these performance differences would be negligible most of the time, but if the compiler was to be used for an enterprise entity, these differences might be more clearly seen.

### Omitting the Visitor Pattern

Most recursive descent compilers make use of the Visitor Pattern for evaluating and executing compilation. Methods in this compiler, such as evaluate(), compile(), and others are all included in the parser directly, rather than being in its own class that's only referenced. The trade off with this design choice is simplicity at the cost of organization and the volume of code. By writing each method into its respective parser class, writing the compiler was much easier than it otherwise would be. However, this resulted in some unnecessary bloat to the project which could very well be trimmed by making use of design patterns and optimization.

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

The development lifecycle seen with this project is Test Driven Development (TDD). This design traditionally consists of tests being written as oracles for a project's intended functionality. In the case of this course, the tests were pre-written by the instructor. With these tests present, our development cycle consisted of writing code, testing it against the provided tests, and then fixing any issues that arise due to the results of those tests. This model is largely helpful to a classroom environment, but also provides some challenges when trying to emulate design and development practice in a project. TDD made it extremely easy to focus on the development of the project without having to create test cases or look at node coverage for a given branch of

the project, which cuts down on time significantly. Rather than having to meticulously test each technical aspect of a given piece of code, TDD with provided tests allows us to focus much more of our time on actually developing a project. The main drawback of using TDD in place of some other more elaborate testing is that there will inevitably be some cases that are not fulfilled. An example of this is with the assignment statements in our parser, which was never tested by the provided tests. By using TDD, there will be some components of a project that might just not work, or might have some drastic side effects that are not intended. Since this project will have no greater use beyond education, there's no need to worry about potentially dangerous bugs, but in a real environment, using TDD on a compiler might be a dangerous idea unless the objective was to make a functional compiler in a short amount of time.