

Montana State University

# CSCI 468: Compilers

Senior Capstone Portfolio

Megan Fehres, Madison Munro  
5-5-2023

## Section 1: Program

To view the full code for the Catscript capstone project including the source code, testing suites, and helper classes, please see **/capstone/portfolio/src.zip**. The source code is listed under **src/main** and the test suites are listed under **src/test** within the zip file.

## Section 2: Teamwork

Each team member worked on their own version of the Catscript coding project, which included the following sections:

1. *Tokenization*: Reading in a string of potential Catscript code and chunking out string elements into lexemes to be matched with a set of tokens (ex: the lexeme “for” will be tokenized as a for loop keyword). This stage also determines where a token is located in the Catscript source code (line number, offset, etc.).
2. *Parsing*: After tokenizing inputted Catscript code, we then use those tokens to construct a grammar for Catscript, and then create a syntax tree based on that grammar.
3. *Evaluation*: Once a grammar and syntax tree are constructed for Catscript, we implement the computations to be performed by Catscript (ex: evaluating the expression “1+1” as “2”).
4. *Bytecode Compilation*: This is another way of evaluating Catscript code, but this time using bytecode to do the compilation and execution.

While most of the functionality of Catscript was programmed by each team member, the basic skeleton of the code was provided to each team member by the capstone instructor, including most of the testing suites, helper functions, abstract classes, and interfaces.

Once the source code was fully programmed by each respective team member (1 and 2), the code was swapped with each other so that the second team member would document, debug, and test the functionality of the Catscript programming language developed by the first, and vice versa.

Code debugging was done via IntelliJ’s built-in debugging functionality.

In addition to providing documentation of the Catscript language, each partner also wrote a test suite with 3 tests to verify that Catscript code evaluation was working properly (see **src.zip/src/test/java/edu.montana.csci.csci468/demo/CatscriptCapstonePortfolioTest.java**). The test suite found in the submitted zip file was done by team member 1 for team member 2 while the test suite done by team member 2 for team member 1 is located

in that member's zip file submission. Further testing was also provided by test suites created by the capstone course instructor.

To see the full documentation of the Catscript language, please refer to **Section 4: Catscript Guide**. The guide provided in this portfolio was done by team member 1 for team member 2 while the documentation done by team member 2 for team member 1 is in team member 1's portfolio.

Each team member spent an equal amount of time and work on the capstone project, totaling to about 90 hours on Catscript software development (75% of the total Catscript work) about 25 hours on code debugging, documentation, and testing (21% of the total Catscript work), and about 5 hours on authoring our respective portfolios (4% of the total Catscript).

## Section 3: Design Pattern

The design pattern used and implemented in the Catscript capstone project was the **Memoization** pattern. Implementation of this design pattern is located in the Java file *CatscriptType* found in `src.zip/src/main/java/edu.montana.csci.csci468/parser/CatscriptType.java`. This pattern was used in the project to improve the efficiency of the *getListType()* function and reduce the number of objects created when creating new list types. With Memoization, only one list type is created for each corresponding Catscript type, which eliminates the need for redundant object creation. Thus, the program speed is improved, and memory usage is reduced significantly. If this function were not implemented with Memoization, the constructor used to create a new list type could be expensive to use hundreds of times when executed in addition to taking up more storage than needed and slowing the program down.

The function is shown below. A HashMap containing the list types for a given Catscript type is created, and then it is used to determine if a list type object was already created or not. If it wasn't, a new list type is created, otherwise you return the type created from a previous code execution:

```
private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new
HashMap<>();

public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = LIST_TYPES.get(type);
    if(listType == null){
        listType = new ListType(type);
        LIST_TYPES.put(type, listType);
    }
}
```

```
    return listType;  
}
```

## Section 4: Catscript Guide

### Introduction:

Catscript is a simple scripting language. Here is an example:

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

What was shown above was Catscript's way of defining a variable and then printing out the value associated with the variable, which will print out "foo." This is one of the many features of Catscript.

Below are said features of the Catscript programming language. Many of them are simple versions of the staple features found in other languages.

### Features:

#### Catscript Types:

Catscript features a simple type system including:

- int: for integer values
- string: for string values
- bool: for true/false values
- object: for object values
- void: no type
- null: for empty values
- list<>: for list of types available

Note that there is no floating-point type in Catscript. This is because Catscript is meant to be a simple language used for learning purposes.

#### Primary Expressions:

Primary expressions in Catscript are as follows:

- Strings

- Integers
- Booleans
- Null values
- Parenthesized expressions (or parenthesis in general)
- Lists
- Function calls
- Identifiers

### Unary Expressions:

Unary expressions are used in Catscript to denote a value as the opposite of what it normally evaluates to, or to set an integer value to be negative. For example, the following code chunk:

```
not true
-1
```

Evaluates the first expression as “false” and the bottom expression as a negative 1. Note that the “-” symbol only works for integer values and the “not” keyword only works for boolean values.

### Factor Expressions:

Factor expressions In Catscript are expressions where two or more integer values are either multiplied or divided. Here are some examples:

```
2 * 2
10 / 5
2 * 4 / 2
10 / 2 * 7
```

Note that factor expressions can only be applied to integer types in Catscript.

### Additive Expressions:

Additive expressions in Catscript are expressions where two or more values are either added or subtracted. Here are some examples:

```
2 + 2
10 - 5
2 + 4 - 2
10 - 2 + 7
```

In addition, you can use the “+” symbol for string concatenation in Catscript. Here are some examples:

```
"I am " + 10 + " years old!"  
"a" + null
```

For string concatenation, as long as one of the evaluated sides is a string, the elements concatenated together will become one large string, even if a value is null or an integer.

Apart from string concatenation, the additive expression can only be used on integers (you can't add/subtract boolean or string values).

### Comparison Expressions:

Comparative expressions in Catscript are used to compare integer values to each other in the following ways:

```
int1 < int2  
int1 > int2  
int1 <= int2  
int1 >= int2
```

### Equality Expressions:

Equality expressions in Catscript are used to verify if one value is or isn't equal to another, which can be applied to integers, booleans, and strings. Here are examples for both equal and not equal expressions:

```
true == true  
1 != 2  
"a" == "a"
```

### For loops:

For loops in Catscript are coded/executed like any other language where you loop through a block of code for a certain number of times. In Catscript, statements in for loops are executed by looping through objects in a list, which can be used to set an amount of loop iterations. Here is an example of Catscript's for loop:

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

This demonstrates iterating through a list in Catscript. When executed, each element of the list should be printed out. Here is another example:

```
var y = "Hello!"  
  
for(i in [1,2,3,4]){
```

```
    print(y)
}
```

This demonstrates how you can iterate over a set number of times to execute a chunk of code. The code should print out “Hello!” 4 times.

### If statements:

If statements in Catscript behave much like any other programming language, but it is simplified to just if/else (no else if, elif, etc....). Here is an example:

```
var x = 5

if(x == 5){
    print("x is five")
} else {
    print("x is something else")
}
```

Here, we have a variable, x, set to 5. Depending on what value x is set to, we will execute a different print statement based on if the first condition of the if statement is true or false.

### Printing:

To print out values in Catscript, all one needs to do is to type out “print(thingToPrint).” Here are some examples:

```
print(12)
print("Bye")

var x = "This is a line"
print(x)
```

When executed, the first two print statements print out “12” and “Bye” back to the user. The last print statement shows how to print out a value stored in a variable.

### Variables:

There are a few different ways to define variables in Catscript. One example is as follows:

```
var x = 10
```

Here, the keyword “var” is used to declare a variable called “x,” which has the value 10 assigned to it. You can call back to this variable as such (which an assignment statement):

```
if(x==10){
    x = 11
}
```

```
}  
print(x)
```

This example shows that we can refer back to an already defined variable and assign it another value (setting x to 11 if x is 10).

Now, Catscript has the feature where it infers the type of the variable based on what is assigned to it. For example, setting a variable to the integer value 10 will tell Catscript that this variable is an integer type, even without explicitly declaring it as such. However, you can explicitly define variables to be of a certain type in Catscript. Here is how to do it:

```
var x : int = 10  
var y : bool = false  
var x : string = "asdf"
```

As seen in the example above, you explicitly set the type of a variable by adding a colon and then a type supported in Catscript after defining a variable and before setting a value to it. This tells Catscript how to treat the variable defined.

In addition, you can set variables to be list types:

```
var x : list<int> = [10,11,12]
```

Lists can also be defined without a component type:

```
var x : list = [10,11,12]
```

## Functions:

Functions in Catscript are self-explanatory. The bare minimum to define a function in this language is to use the “function” keyword as such:

```
function foo() {}
```

This is an example of a void function with no arguments and no statement body. Arguments can be passed in with or without explicit types set. Here are examples for both:

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

You can also pass in multiple arguments with or without explicit types:



```
function foo(x,y,z) {
    print(x)
    print(y)
    print(z)
}

function bar(x : string, y : int, z : string) {
    var s : string = x + " " + y + " " + z
    print(s)
}
```

To call a function after it is defined, you just call the name of the function as so:

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

### Return Statements:

Return statements are used in Catscript whenever a function needs to return back a value of a certain type. To set up a return value, you must define a function to be of a certain type:

```
function foo(x : int) : int {
    return x
}
```

By adding a semicolon and a type after your arguments are defined in a function definition, you are telling Catscript what value you want returned from the function defined. If the function type is void, you do not need a return statement.

### Extra Notes:

As Catscript is a scripting language, there is no support implemented for Object-Oriented Programming (i.e., there is no support for defining classes or interfaces). There is also no implementation of “try-catch” or other error handling techniques.

Catscript is also a statically typed language, meaning that variable types are known at compile time. Consider the following code:

```
var x = "asdf"

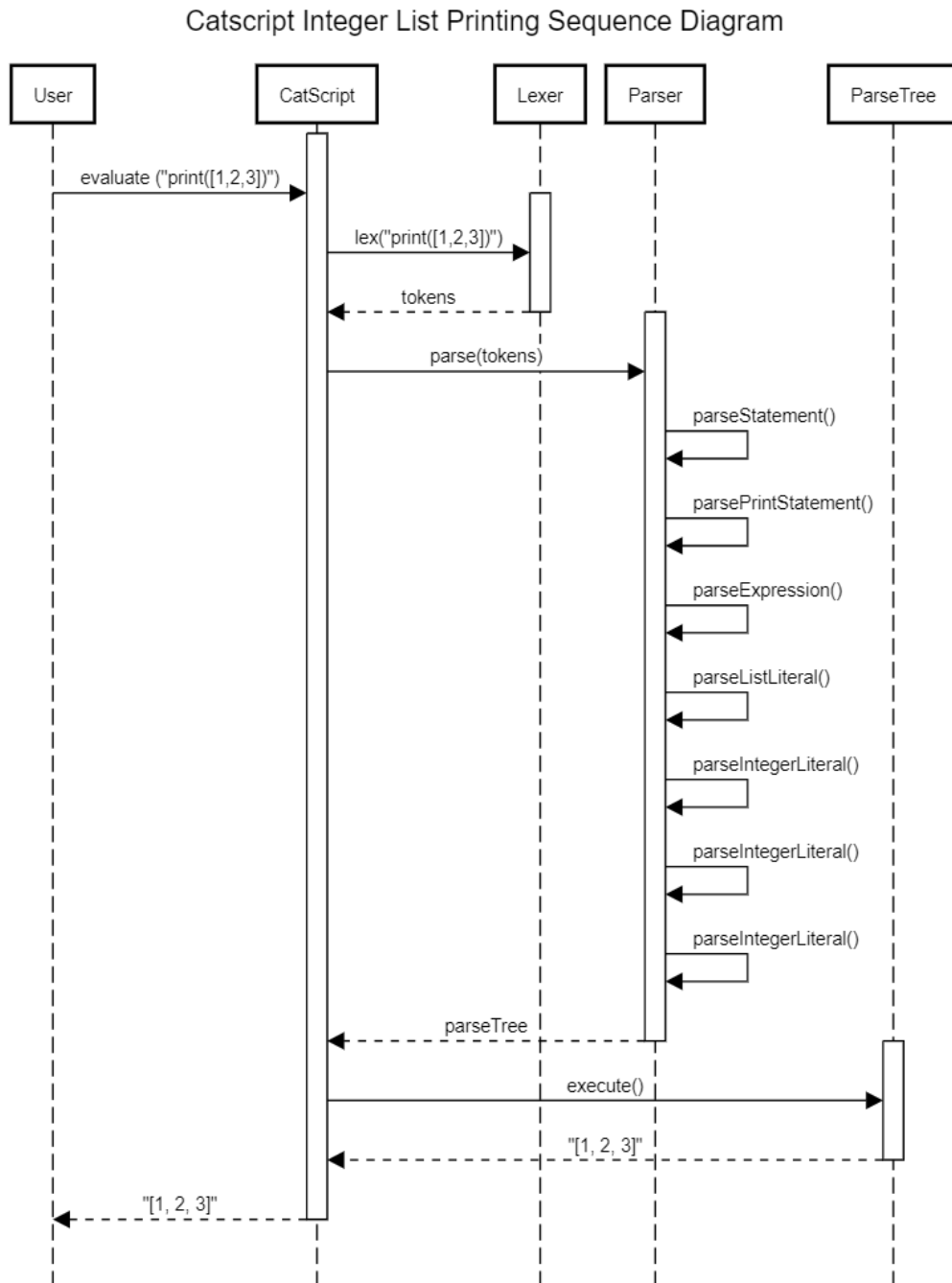
function add(i : int) : int {
    return i + i
}

print(add(x))
```

This will result in a compilation error because the type for “x” is statically known as a string type, not an int type, so it can’t be passed into an integer type function.

## Section 5: UML

The following visual is a Sequence Diagram generated to show the concept of how statement and expression evaluation would be handled in Catscript, specifically how printing a list literal would be evaluated. This was made after each team member coded parsing functionality but before each team member coded evaluation functionality:



*Catscript Evaluation Sequence Diagram*

## Section 6: Design Trade-Offs

One design trade-off decision made while developing Catscript was that both team members programmed their Catscript parser by hand. This was done via a Recursive-Descent algorithm. We chose this because we felt it was more intuitive to implement during each team member's development of the language, and that we wouldn't have had a better understanding of how grammars and parsing worked if we had used a parsing tool such as LEX or ANTLR. One drawback we found with using recursive descent is that each team member had to write more code than usual, which using a parsing tool would've solved for us since all we would've had to do in that case is create a lexer grammar in a separate file to be read into the code using TLexer. However, the parser code that is generated by this lexer is complicated to understand and interpret compared to implementing by hand, which is why each team member decided to use Recursive-Descent rather than a parsing tool. Below is how Recursive-Descent parsing works:

Recursive-Descent is where you parse and evaluate expressions in order of precedence. The order of precedence (from low to high) is as follows:

- Equality Expressions (`==` or `!=`)
- Comparison Expressions (`<`, `>`, `<=`, `>=`)
- Additive Expressions (`+` or `-`)
- Factor Expressions (`*` or `/`)
- Unary Expressions (`!` or `not`)
- Primitive Expressions (identifiers, strings, integers, parenthesis, lists, etc.)

With Recursive-Descent, you start by parsing/evaluating equality expressions (if there) on the left-hand side, and then move on to the next highest precedence expression after evaluating the right-hand side of the expression.

Statement parsing/evaluation is somewhat similar except we look for specific keywords when parsing, and then see what comes after in to determine if it is a legal statement to make in Catscript. Unlike with expressions, we're not moving down a hierarchy of precedence, rather we are searching for parsed tokens that match the grammar required for said statement.

## Section 7: Software Development Life Cycle Model

Throughout the capstone project, Test Driven Development (TDD) was used as the model for Catscript's development. As mentioned in **Section 2**, this was accomplished via test suites created by either the instructor or by each respective team member. This model helped each team member out because the tests were written and designed in a way that

will tell you exactly where your code fails when testing certain cases, which provides a clear direction for each team member to go to in the code when debugging for errors. These tests were designed to make sure that the functionality of Catscript worked properly as a scripting language.