

# **CSCI-468: Compilers Capstone**

Spring 2023

Hank Breckenridge

Partner: Patrick Tung

## **Section 1: Program**

The zip file for the compiler I wrote for this project can be found in this direction. The file is named “source.zip”.

## Section 2: Teamwork

This project had a good balance between individual accountability and real-world teamwork. I was responsible for implementing the functionality of the entire compiler. This work was split into four sections: the tokenizer, the parser, the evaluation of Catscript, and the generation of JVM bytecode. My partner acted as both the documentation and test engineer for my project. In turn, I acted as his project's documentation and test engineer. This ensured that each of us was responsible for implementing a compiler, but it also allowed us to simulate a real-world environment where collaboration is necessary.

The documentation that each of us wrote outlined the complete functionality of the Catscript programming language. It was modeled on Catscript's grammar, which ensured total coverage of the feature set. Since the project already had complete coverage at the unit test level, my partner and I were responsible for writing higher-level tests that acted as integration or system tests between the separate units.

The compiler implementation took the majority of the time, so I spent roughly 80% of the time working on my implementation and roughly 20% of the time writing documentation and tests for my partner. Since each of us had an individual submission the work was split 50/50.

## Section 3: Design Pattern

The main design pattern used in the compiler was memoization. In this design pattern, we implement a sort of caching that allows us to reuse data structures rather than instantiating a new one each time the “getListType” method is called. This pattern is a useful optimization in our program as it eliminates the unnecessary overhead inherent to object instantiation. While this optimization was not needed for the small examples we used in the test cases, larger programs will certainly benefit from this optimization. The code I used to implement the memoization pattern is shown below.

```
public 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: Technical Writing

---

## Introduction

---

Catscript is a simple scripting language.

Here's an example of Catscript in action:

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

## Table of Contents

---

### [Statements](#)

- [For Loop Statement](#)
- [If Statement](#)
- [Print Statement](#)
- [Variable Statement](#)
- [Assignment Statement](#)
- [Function Call Statement](#)

### [Function Declaration](#)

- [Function Body](#)
- [Parameter List](#)
- [Return Statement](#)

### [Expressions](#)

- [Equality Expression](#)

- [Comparison Expression](#)
- [Additive Expression](#)
- [Factor Expression](#)
- [Unary Expression](#)
- [Primary Expression](#)
- [List Literal Expression](#)
- [Function Call Expression](#)
- [Argument List Expression](#)
- [Type Expression](#)

## Features

---

- Catscript includes all the fundamental features of a programming language.
  - These features can be categorized as **statements** or **expressions**.
  - **Statements** execute actions or control the code flow, while **expressions** evaluate to a single value.
  - **Function declaration statements** are more complex and consist of several other components, so they are listed separately from Statements.
- 

## Statements

---

### For Loop Statement

To declare a for loop in Catscript, use the “for” keyword followed by a set of parentheses containing an identifier, the “in” keyword, and an expression that produces a list of values. This expression can be created using any code that generates a list of values, such as a list literal or a function call that returns a list. Once the expression is evaluated and produces a list, the for loop executes a sequence of statements enclosed within curly braces for each item in the list.

Note that in Catscript, you don’t need to declare the identifier before using it in the for loop.

Here's an example of how to use a for loop in Catscript to print the numbers 1 to 5:

```
for (i in [2,3,4,5,6]) {  
  print(i)}
```

### **If Statement**

In Catscript, you can declare an if statement using the “if” keyword followed by a set of parentheses that contain a boolean expression. If the expression evaluates to true, the statements enclosed within the curly braces following the if statement will be executed. Optionally, you can include an “else” clause that contains statements to be executed if the expression evaluates to false.

In the following example, the if statement checks if the value of the variable x is greater than 5, and if it is, the string ‘x is greater than 5’ will be printed.

```
var x = 0  
if (x < 4) {  
  print("x is less than 4")}
```

### **Print Statement**

In Catscript, you can use a print statement to write a value to the output. To declare a print statement, use the “print” keyword followed by a set of parentheses containing an expression to be printed. This expression can be a variable, a literal value, or the result of an expression.

Keep in mind that the string representation of the expression will be written to output, which means that print statements in Catscript are not limited to printing strings.

In the following example, the print statement writes the string ‘Hello, world!’ to the output:

```
print("Carson Gross is a cool instructor.")  
  
// Carson Gross is a cool instructor.
```

In the following example, the print statement displays the result of a calculation:

```
print(9 + 1)
```

```
// Output: 10
```

## Variable Statement

To declare a variable and assign it an initial value in Catscript, you can use a variable statement. The statement is declared using the “var” keyword, followed by an identifier, an optional type annotation, an equal sign, and an expression. This expression can be a literal value, the result of an expression, or the return value of a function call.

Remember that the type annotation in the variable statement is optional, but it can be used to specify the type of the variable.

In the following example, a variable named “y” is declared with an initial value of the result of a calculation:

```
var y = 7 + 2
```

In the following example, a variable named “name” is declared with an initial value of “Hank” and a type annotation of “string”:

```
var name: string = "Hank"
```

## Assignment Statement

In Catscript, you can use an assignment statement to modify the value of a previously declared variable. To declare an assignment statement, use an identifier followed by an equal sign and an expression that evaluates to the new value of the variable.

Keep in mind that for an assignment statement to work in Catscript, the variable must have been previously declared, and the type of the expression on the right-hand side of the equals sign must be compatible with the type declared for the variable.

In the following example, the variable x is declared with an initial value of 10, and then its value is changed to 5 using an assignment statement:



```
var x = 8
x = 3
print(x)

// Output: 3
```

## Function Call Statement

In Catscript, you can use a function call statement to invoke a function and execute its code. The statement consists of the function name followed by parentheses containing any arguments passed to the function. Remember that the arguments passed to the function must match the parameters defined in the function's declaration in terms of data type and order.

Here's an example of a function call statement, where the function `myFunction` is called with three arguments:

```
myFunc(4, "Express", false)
```

For a more detailed explanation of the function call syntax in Catscript, please refer to the [Function Call Expression Section](#).

## Function Declaration Statement

---

In Catscript, you can use function declaration to define a reusable piece of code that can be called from different parts of the program. To declare a function, use the “function” keyword followed by the function name, a set of parentheses that contain a list of parameters (if any), and the function's body enclosed in curly braces.

Here's the basic syntax for a function declaration statement:

```
function subtract(num1, num2): int {
  var result = num1 - num2 return result}
```

Next, let's take a look at the three types of statements that can be used in function declaration statements in Catscript.

## Function Body

The function body is where the actual logic of the function resides, and it can contain any valid Catscript statement, such as variable declarations, control structures, and function calls. The function body is defined within the curly braces {} that follow the function declaration statement.

Keep in mind that the function body statement in Catscript must contain at least one Catscript statement.

The following example illustrates a valid function body statement:

```
function myFunc(x: int): bool {  
  var flag = false if (x <= 8) { print("x is less than or equal to 8") flag = t
```

## Parameter List

The parameter list is a comma-separated list of parameters that the function accepts, with each parameter consisting of a name and an optional type annotation. If a type annotation is not provided, Catscript will try to infer the type from the value passed in when the function is called.

Here's an example of a parameter list composed of two values with type annotations:

```
function myFunc(param1: bool, param2: string) {  
  // function body}
```

## Return Statement

In Catscript, a function can return a value using the return statement. If the function returns a value, the return type must be specified in the function declaration. The return statement can also be used to exit a function early and return a value.

Keep in mind that in Catscript, all branches of code are expected to have a return statement.

The following example illustrates a function with valid return coverage.

```
function voidFunc(param1: string, param2: string): string {  
    return param1 + param2}
```

A function with invalid return coverage has at least one branch of code that will never hit a return statement.

Here's an example of a function with invalid return coverage:

```
function invalidFunc(x: int): bool {  
    if (x >= 7) { return true } else { print("Less than 7") }}
```

If an argument less than 5 is passed into the function, the return statement will never be executed, resulting in invalid return coverage.

## Expressions

### Equality Expression

In Catscript, you can use equality expressions to compare two values and determine if they are equal. The `==` operator is used to check if two values are equal, while `!=` is used to check if two values are not equal. Equality expressions evaluate to boolean values and are frequently used as the expression in if statements.

Here's an example that compares two numbers using the equality expression:

```
if (1 == 0) {  
    print("0 and 1 are equal")  
} else {  
    print("0 and 1 are not equal")  
}
```

### Comparison Expression

Comparison expressions can be used to compare values and produce boolean results. Available operators include:

`>` for greater than

`<` for less than

`>=` for greater than or equal to

`<=` for less than or equal to

```
if (0 > 1) {  
  print("x is greater than y")  
}  
else {  
  print("x is not greater than y")  
}
```

## Additive Expression

Catscript provides additive expressions to perform arithmetic operations on numeric values. The addition operator `+` is used to add two values, while the subtraction operator `-` is used to subtract one value from another.

Here's an example of a print statement that uses an additive expression:

```
print(2 + 3) // Output: 5
```

Additive expressions can also be used with string values. In this case, the addition operator `+` is used to concatenate strings together. The following example demonstrates this behavior:

```
var str1 = "HELLO"  
var str2 = "WORLD!"  
print(str1 + ", " + str2) // Output: HELLO, WORLD!
```

## Factor Expression

In Catscript, a factor expression refers to an expression that involves multiplication or division of two operands. The order of operations applies to factor expressions just like any other mathematical expression. Multiplication and division are performed before addition and subtraction, so it's important to use parentheses when necessary to ensure that the expressions are evaluated in the correct order.

The following example demonstrates a simple multiplication operation between two integers:

```
print(3 * 11) // Output: 33
```

Here's an example of a factor expression that includes parentheses to enforce order of operations with an additive expression:

```
var result = (2 + 1) / 3  
  
print(result) // Output: 1
```

## Unary Expression

A unary expression is an expression that operates on a single operand. The unary operator can be either `not` or `-`.

The `not` operator performs a logical negation on a boolean value. When applied to a boolean value, it returns the opposite boolean value. The following example demonstrates this:

```
var y = false  
print(not y) // Output: true
```

The `-` operator in Catscript performs negation on a numeric value. When applied to a numeric value, it returns the value multiplied by `-1`. Here's an example:

```
var y = 3  
print(-y) // Output: -3
```

## Primary Expression

In Catscript, a primary expression is the simplest form of expression and can take on several forms, such as an identifier, a literal, a function call, or a parenthesized expression.

Identifiers are used to access the value of a variable or to call a function. In the following example, `name` is an identifier:

```
var name = "Johnson"
```

Literals are fixed values that are directly represented in the code. There are several types of literals, including strings, integers, booleans, null, and list literals.

```
"Hello, World!" // String literal
3               // Integer literal
false          // Boolean literal
null           // Null literal
[2,3,4]         // List literal
```

Function calls are used to invoke a function with zero or more arguments. A function call is denoted by the function name followed by parentheses containing the arguments passed to the function.

```
function greet(name) {
  print("Greetings, " + name + "!")
}

greet("Johnson")
// example Function call
```

Parenthesized expressions are used to group expressions together to enforce order of operations or to clarify code. The expression inside the parentheses can be any valid expression.

```
30 / (9 + 1)
```

## List Literal Expression

In Catscript, a list is a collection of ordered values. A list literal is a way to define a list with a specific set of values. List literals are enclosed in square brackets `[]`, with each element separated by a comma `,`.

Here's an example of a list literal containing three integers:

```
[7,9,0]
```

List literals can also include elements of distinct types:

```
[false, null, 5, "cat"]
```

## Function Call Expression

A function call expression is used to call a function and pass arguments to it. It consists of the function name followed by parentheses containing the arguments passed to the function. The arguments can be expressions that evaluate to the expected data types specified in the function's parameter list.

It's important to note that in Catscript, the function being called must have been previously defined through a function declaration statement.

The syntax for a function call expression in Catscript is as follows:

```
func(arg1, arg2, ..., argN)
```

Here's an example of a function call expression in Catscript that passes two arguments, an integer and a string, to a function called `myFunction`:

```
myFunc(1, "Hi")
```

Note that the order and number of the arguments passed to the function must match the parameters defined in the function's declaration. It's also worth noting that the function call expression is an expression that evaluates to a value. The value returned by the function can be stored in a variable or used in an expression.

## Argument List Expression

Catscript uses an argument list expression to pass one or more arguments to a function. This expression consists of zero or more expressions enclosed in parentheses and separated by commas.

The example below shows an argument list being used in a function call, with the argument list being the series of expressions between the parentheses.

```
myFunc(6, "yes")
```

## Type Expression

CatScript variables are statically typed, which means their type is determined at compile-time and cannot change during program execution. Type expressions are used in function declarations, variable statements, and parameter lists. The CatScript type system includes the following types:

- `int` : A 32-bit integer.
- `string` : A Java-style string.
- `bool` : A boolean value, which can be either true or false.
- `list<x>` : A list of values with a specific type `x`.
- `null` : The null type, which represents the absence of a value.
- `object` : The most general type, which can hold any kind of value.

The following example shows a variable statement with a type expression:

```
var num: int = 0
```

Type expressions can be used to indicate the return type of functions and specify the parameter types, as shown in this example:

```
function isLessThan(x: int, y: int): bool {  
    return x < y  
}
```



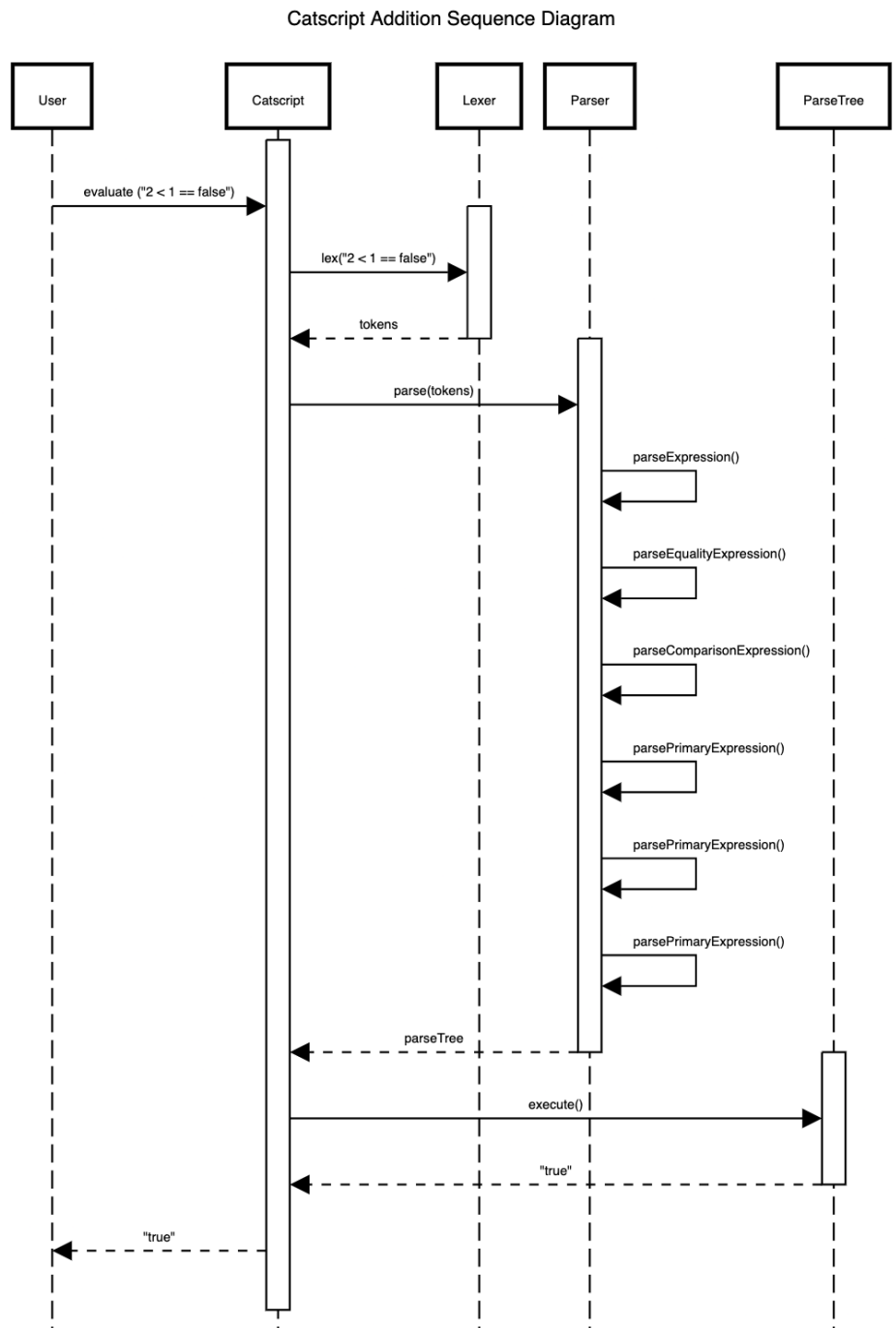
## Section 5: UML

The following sequence diagram demonstrates how each component interacts to evaluate a Catscript program. In this case, the input is a fairly simple equality expression.

As you can see, the lexer breaks the expression up into individual tokens that are returned to the Catscript program.

The parser then uses the tokens to build a parse tree via recursive descent. The recursive nature of the parser can be seen in the chain of parse method calls. Each call becomes more specific from the top-level “parseExpression()” call to the three “parsePrimaryExpression()” calls (two for the integers in the comparison expression and one for the boolean literal).

Once the parse tree is constructed, the execute method is called, and we can see that the value of true is passed back to the user.



## Section 6: Design Trade-offs

One design tradeoff in this project was implementing the evaluate method on the parse tree nodes instead of utilizing the visitor pattern. The visitor pattern is very common in recursive descent parsing, but it leads to a lot of complexity in the codebase. The main value of the visitor pattern is improved code cleanliness which makes changing or extending the codebase much more simple. The visitor pattern decouples the recursive descent algorithm from the parse tree node data structure. This makes the life of the developer easier when changing the codebase in the future.

Since we are not designing Catscript to be maintained for a long time, using the visitor pattern does not make sense. Instead, I implemented the evaluate method directly on the parse tree node data structure. As described earlier, this comes with a number of downsides. The codebase becomes much more rigid due to the tight coupling between the recursive descent algorithm and the parse tree data structure. This is not a significant consideration in our case because we do not have to maintain the codebase after satisfying the specification. Therefore, it makes sense to favor simplicity over flexibility and extensibility in this case.

## Section 7: Software Development Lifecycle

The software development lifecycle model used in this project was the test-driven development (TDD) model. In this model, we are given a set of tests that outline the expected functionality of the program. These tests all failed in the beginning but served as a guide for implementing the compiler. I found this model very useful as it quantifies exactly what the program needs to do.

I have used a similar model (design-by-contract) in a recent project, and I noticed a lot of similarities. Most notably, specifying exactly what the program must do to satisfy all requirements was very difficult. Fortunately, Professor Gross did the heavy lifting of providing the test suites ahead of time. This allowed me to move very quickly from feature to feature without having to worry much about whether or not the edge cases were covered.

I have also used the agile model in internships and personal projects. This model was great in that it did not have the high up-front cost and allowed my team to change the requirements as the project developed, but the lack of structure compared to TDD made it much more difficult to quantify the expected behavior of the program.