



CSCI 468 Compilers

Capstone Portfolio

May 5, 2023

Megan Fehres

Madison Munro

Section 1: Program

To view the complete codebase 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

During the development of the Catscript coding project, each team member was responsible for working on various sections of the project, including the critical components of tokenization, parsing, evaluation, and bytecode compilation. The first stage, tokenization, involved analysing a string of potential Catscript code and extracting the individual string elements to be matched with a set of predefined tokens. This process also determined the location of each token within the Catscript source code, such as its line number and offset. Following tokenization, we moved on to the parsing stage, where we used the extracted tokens to create a grammar for Catscript and then constructed a syntax tree based on that grammar. In the evaluation stage, we implemented the computations necessary for Catscript to operate, such as evaluating the expression “1+1” to arrive at a result of “2”. Finally, in the bytecode compilation stage, we utilised bytecode to compile and execute Catscript code, enabling another means of code evaluation. Each team member worked independently on their version of the project, taking charge of various project stages to ensure its success. 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.

The source code generation was done on an individual level; my partner and I were responsible for creating each of the four parts of the compiler, so contributed to each other's project by meeting up for the purpose of debugging, testing, validation, and putting together the documentation. While drafting the documentation, each partner created a test suite with three tests for the other for the purpose of validating each other's compiler (see [src.zip/src/test/java/edu.montana.csci.csci468/demo/CatscriptPortfolioTests.java](#)). To see the full documentation, refer to section 4. The Catscript documentation presented in this portfolio was done by team member 1 for team member 2.

Total Estimated Time: 115 Hours

Team Member 1:

- Software Developer
- Total Estimated Time: 90 hours, totaling 75% of the work

Team Member 2:

- Technical Documentation, Testing, Debugging
- Total Estimated Time: 25 hours, totaling 21% of the work

The portfolio itself took 5 hours to complete, totaling 4% of the work.

Section 3: Design Pattern

Catscript uses the **Memoization** Pattern as its primary design pattern. This pattern is used in the CatscriptType class to improve the efficiency of the getListType function and reduce the number of objects created. With **Memoization**, only one ListType is created for each corresponding CatscriptType, which eliminates the need for redundant object creation. This results in improved program speed and reduced memory usage. If this function were to be implemented in a simple way, it would be necessary to create and store a new list type, as well as carry out extraneous comparison operations to determine which list type to generate.

This design pattern is found in the CatscriptType class in the getListTypeFunction, located at csci-468-spring2023-private/src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java. The pattern affects the entire function and the way it is structured, compromising the function and its control flow. It uses a static final HashMap to store CatscriptTypes along with their corresponding ListType. This is what allows for improved program speed, reduced memory usage, and the efficient retrieval of ListTypes without creating new objects each time as mentioned above.

```
static final HashMap<CatscriptType, ListType> CACHE= new HashMap<>();

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



Section 4: Technical Writing

Introduction:

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

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

This is Catscript's way of defining a variable and printing out the value associated with it; this will print "foo." This is one of many features in Catscript.

Listed below are the features of the Catscript programming language, many of which are simple versions of the principal features found in other languages.

Features:

Catscript Types:

Catscript features a fairly simple type system:

- 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

Keep in mind, there is no floating point type in Catscript, since it is meant to be a simple language used for educational purposes.

Catscript is a statically typed language, meaning the type of a variable is determined at compile-time rather than at runtime. The data type of a variable must be explicitly declared before it can be used in the program. Consider the following code:

```
var num = 2023  
Function concat(str: string): string {  
    return str + str  
}
```

```
print(concat(num))
```

The variable `num` is declared without specifying its type. The compiler would infer that `num` is a number type based on the assigned value `2023`. However, in the function `concat`, the parameter `str` is explicitly declared as a string type. When `num` is passed as an argument to `concat`, it is incompatible with the expected string type, and the compiler would raise a type error during compilation. The code would not compile.

Primary Expressions:

Primary expressions in Catscript are as follows:

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

Comparison Expressions:

Comparative expressions in Catscript are used to compare integer values to each other. For example:

```
num1 <= num2
num1 >= num2
num1 < num2
num1 > num2
```

Equality Expressions:

Equality expressions in Catscript are used to verify if a value is or is not equal to another. This can be applied to integers, booleans, and strings. For example, for both equal and not equal expressions:

```
false == false
"answer" == "answer"
10 != 100
```

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 negative. For example:

```
not true  
-1
```

Evaluates the first expression as “false” and the bottom expression as a negative 1. Keep in mind, 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. For example:

```
2 * 2  
100 / 20  
6 * 4 / 8  
15 / 3 * 2
```

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. For example:

```
2 + 2  
40 - 18  
8 - 4 + 4  
10 + 4 + 7
```

The “+” symbol can be used for string concatenation as well. For example:

```
"I have $" + 200 + " dollars"  
"a" + null
```

Regarding string concatenation, if 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.

With the exception of string concatenation, the additive expression can only be used on integers. This means adding or subtracting boolean or string values will not work.

Variables:

In Catscript, variables are declared using the “var” keyword. For example:

```
var x = "yes"
```

The keyword “var” is used to declare a variable called “x,” which has the value “yes” assigned to it. The variable can be called back to by its given name with an assignment statement:

```
if (x=="yes") {  
    x = "approved"  
}  
print(x)
```

An already defined variable can be referred back to, and we can assign it another value (setting x to “approved” if x is “yes”).

Catscript has a feature that allows it to deduce the type of a variable based on its assigned value. For instance, assigning an integer value of 20 to a variable automatically assigns the integer type to that variable without explicitly declaring it. Nonetheless, Catscript provides an option to explicitly define variables as a specific type:

```
var x : int = 20  
var y : bool = true  
var z : string = "hello"
```

As seen in the example above, after defining a variable and before assigning a value to it, the type can be specified in Catscript by appending a colon followed by a supported type. This instructs Catscript on how to handle the declared variable.

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

```
var x : list<string> = ["one", "two", "three"]
```

Lists can also be defined without a component type:

```
var x : list = ["one", "two", "three"]
```

Printing:

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

```
print("How are you?")  
print("99")
```

```
var x = 8090  
print(x)
```

Upon execution, the first two print statements print "How are you?" and "99". The last print statement shows how a value is stored in a variable and then printed.

Functions:

Functions in Catscript are straightforward to define. At a minimum, you can use the "function" keyword, followed by the function name and a pair of parentheses to create a void function with no argument and no statement body.

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

You can also define arguments with or without explicit types. The following are examples of both:

```
function foo(num) {  
    print(num)  
}  
  
function bar(num: int) {  
    print(num)  
}
```

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


```
function foo(num,truth,str) {
    print(num)
    print(truth)
    print(str)
}

function bar(num : int, truth : bool, str : string) {
    var s : string = num + " " + truth + " " + str
    print(s)
}
```

To call a function after it is defined, you simply call the name of the function:

```
foo(5,true,"answers")
```

Return Statements:

In Catscript, return statements are utilised when a function needs to send a value of a specific type back. To establish a return value, the following steps must be taken while defining a function:

```
function foo(answer : string) : string {
    return string
}
```

When you include a semicolon and specify a data type after defining the arguments of a function in Catscript, you are indicating to the language what type of value should be returned from the function.

If Statements:

If statements in Catscript behave similar to other programming languages, but it is simplified simply to if/else. For example:

```
var x = "yes"

if(x == "yes"){
    print("accepted")
} else {
    print("denied")
}
```

Variable x is set to “yes”. Depending on the value of x, a different print statement will be executed based on if the first condition of the if statement is true or false.

For Loops:

For loops in Catscript are coded and executed similar to other languages; you loop through a block of code for a given number of times. In Catscript, statements within for loops are executed by looping through objects in a list, which can be used as a way to set the number of loop iterations. For example:

```
var list = ["a","b","c"]  
  
for(i in list){  
    print(i)  
}
```

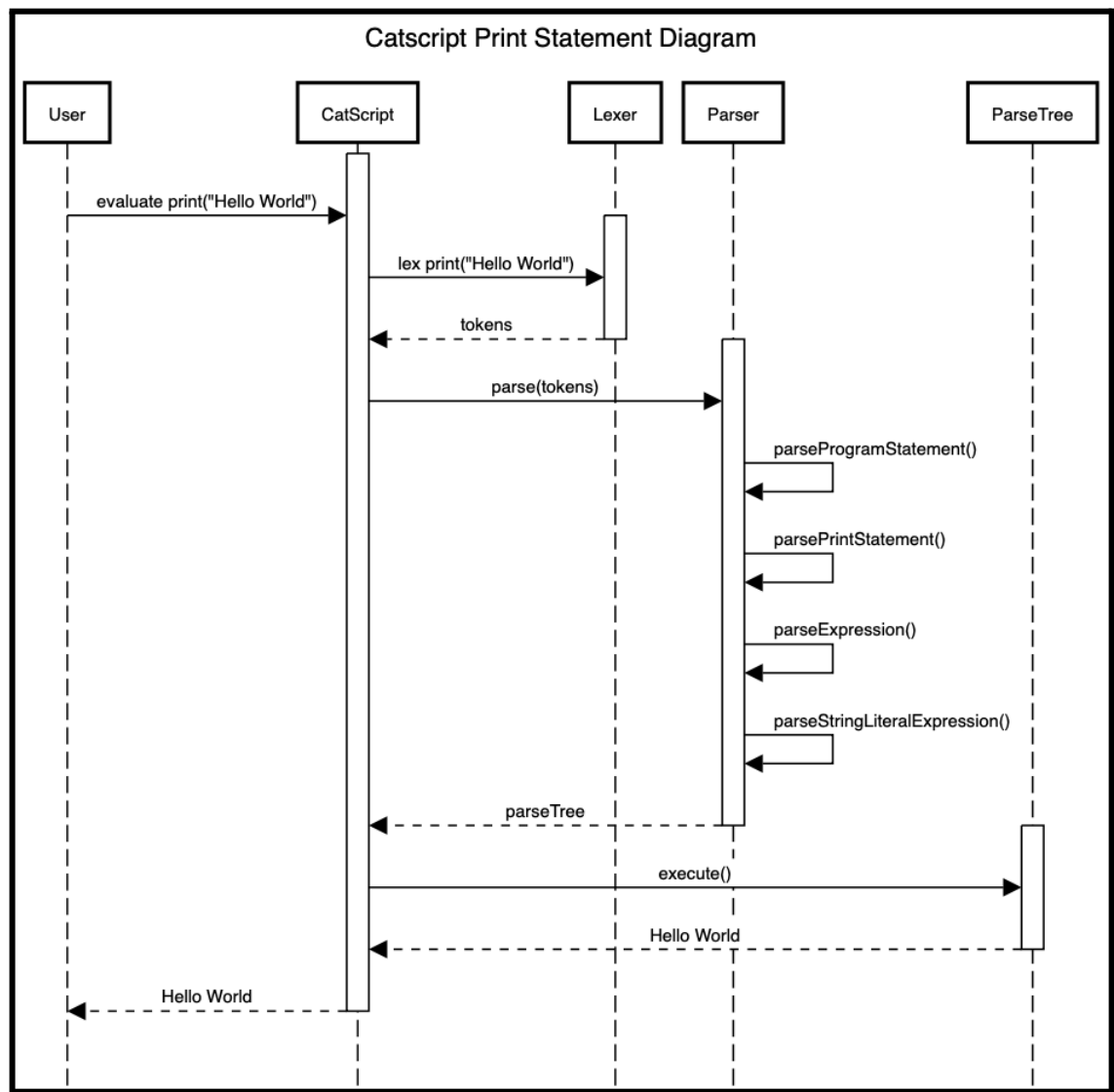
This demonstrates iterating through a list in Catscript. When executed, each element of the list is printed:

```
var y = "Looping..."  
  
for(i in [1,2,3]){  
    print(y)  
}
```

This code chunk demonstrates iterating over a set number of times to execute a chunk of code. The code prints “Looping...” 3 times.

Section 5: UML

The Sequence Diagram below illustrates how statement and expression evaluation would be processed in Catscript, focusing on the evaluation of printing the string "hello world." It was created after the team completed coding the parsing functionality but before implementing the evaluation functionality.



Catscript Evaluation Sequence Diagram

Section 6: Design Trade-Offs

During the development of Catscript, a design decision was made to have both team members manually program the parser using the Recursive-Descent algorithm. This was chosen for its intuitive implementation during each member's development and for better understanding of grammar and parsing. However, this approach required more code writing compared to using a parsing tool like LEX or ANTLR, which would have allowed for creating a lexer grammar in a separate file to be read into the code using TLexer. Despite this drawback, team members found the code generated by a lexer to be more complex to understand and interpret compared to implementing by hand, leading them to stick with Recursive-Descent parsing. In Recursive-Descent parsing, expressions are parsed and evaluated 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.)

In Recursive-Descent parsing, the process starts by evaluating the left-hand side of any equality expressions (if present) and then moves on to the next highest precedence expression after evaluating the right-hand side of the current expression. However, statement parsing/evaluation differs from expression evaluation in that it involves searching for specific keywords in the parsed code and checking whether the tokens that follow are legal statements in Catscript. Unlike expressions, we don't move down a hierarchy of precedence while parsing statements. Instead, we match parsed tokens to the required grammar for a statement.

When compared to other parsing methods like Bottom-Up parsing, Recursive Descent parsing has several advantages and disadvantages. In the case of the Catscript language, which has simpler grammars and structures, Recursive Descent is more suitable. One of the advantages of Recursive Descent is that it is a top-down parsing technique that starts from the highest-level grammar rules and works its way down to the lowest-level rules, making it easier to implement and maintain. Additionally, it offers more flexibility in how the parser is implemented, making it easier to modify and update the parsing algorithm. Finally, Recursive

Descent parsing is generally easier to debug compared to Bottom-Up parsing due to its predictable nature.

Section 7: Software Development Life Cycle Model

During the development of Catscript in the capstone project, we followed the Test Driven Development (TDD) model, which involved creating test suites by either the instructor or team members, as explained in Section 2. This approach was executed through the implementation of test suites created by either the instructor or each respective team member. The TDD model served as a valuable tool for our team members, as it ensured that the test cases were designed and constructed in a meticulous manner that allowed for pinpointing of code failures when specific scenarios were tested. This provided a clear-cut direction for each member to follow when debugging for errors within the codebase. Notably, these tests were designed to rigorously test the functionality of Catscript as a scripting language and ensure that it worked optimally.