

Capstone Report

CSCI-468 COMPILERS

Spring 2023

Jake Rivers & Kate Stallbaumer
Montana State University

Program

The source code of this program can be found at <https://github.com/Kstall23/csci-468-spring2022-private/blob/master/capstone/portfolio/source.zip>.

Teamwork

The project was divided into five distinct sections, with each section accompanied by a set of tests that aligned with the work completed during that specific time frame. Partner 1, Kate Stallbaumer, completed the first four sections independently, using a test-driven development style. However, for the fifth section, Partner 1 and Partner 2, Jake Rivers, collaborated by exchanging new tests that they had developed for any of the content created in the previous sections. Additionally, both partners collaborated on creating the project's documentation. Partner 1 primarily handled documenting the contents and structure of the Catscript Language, while Partner 2 reviewed and made relevant changes. Overall, Partner 1 completed about 95% of the project work, while Partner 2 completed about 5% for this report.

Design Pattern

The major design pattern that was implemented in this project was the memoization pattern. This pattern was used in the CatscriptType.java file (lines 35-46) as a way to prevent redundant initializations of the various list types that can be used when running the compiler. For example, if a list comprised of integers is created, that list type can be stored into the hash map and be quickly retrieved when it is referenced in another location. This speeds up the processing time of retrieving this type reference. The code snippet implementing this design pattern can be seen here:

```
static 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;
}
```

Technical Writing

1 Catscript Guide

1.1 Introduction

Catscript is a small, statically typed scripting language that has comparable syntax to languages like Java and C, although Catscript is much simpler and has fewer features than those full programming languages. Catscript can either be evaluated directly using Java and the Catscript Server or it can be compiled using the Catscript Compiler program and then executed on the Java Virtual Machine (JVM).

1.2 Features

The parts of any Catscript program can be categorized as part of either an expression or a statement. Expressions ultimately evaluate to a value while statements perform an action in the Catscript runtime environment.

1.2.1 Statements

For loops

For loops in Catscript iterate over defined lists and perform a set of operations per iteration. The operations performed are known as the “body” of the for loop. Operations may contain more types of statements such as print- statements and if-statements with no issue Example:

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

Output:

```
1  
2  
3
```

If Statements

If statements in Catscript are able to have conditional statements that restrict access to a portion of code unless the conditional statement is met.

Example with conditional met:

```
if(1 < 2) {  
    print("One is less than two")  
}
```

Example with conditional not met:

```
if(1 > 2) {  
    print("One is greater than two")  
}
```

Print Statements

Print statements in Catscript evaluate an expression and print its value. This can be done with expressions that evaluate to either Strings Literals or Integer Literals.

Example:

```
print("Hello World")
```

Output:

Hello World

Variable Statements

Variable Statements in Catscript used to assign values to variable names. Any combination of valid expressions that evaluate to a value of a single type is assignable to a variable. Types in Catscript can either be stated explicitly after the variable name with a semi colon and a type name or they can be implicitly defined.

Examples:

```
Var x = "Look Mom, I'm a Catscript Variable string value"
```

```
Var y = 1
```

Or:

```
Var x : int = 1
```

Assignment Statements

Assignment statements in Catscript are closely related to variable statements. The assignment statement is used to connect a variable name with a value via an equals sign. It is only possible for variables to be assigned or re-assigned to the same type that they initially defined as, whether they were implicitly or explicitly defined. For example an Integer variable cannot be assigned to a String value.

Example:

```
Var x = 1
```

```
x = 3
```

```
print(x)
```

Output:

3

Function Definition Statements

Catscript utilizes Function Definition Statements to declare functions that can be called from other parts of the program. These functions can have parameters, which can be explicitly or implicitly defined just like variable statements, and must include a body. The body can consist of any type and number of statements. Additionally, functions can contain return statements, which signal the end of the function. When declaring functions that include return statements, it is necessary to specify the return type after the function's name and parameters.

Examples:

```
foo(a, b, c) {  
    print(a)  
    print(b)  
    print(c)  
}
```

Or:

```
foo(a: string, b: string, c: string) {  
    print(a + b + c)  
}
```

Or:

```
foo(a: int, b: int, c: int) : int {  
    return a * b * c  
}
```

Return Statements

In Catscript, Return Statements enable the passing of values from the inner scope of functions to the outer scope where the functions were invoked. These statements terminate the function they belong to, and may contain expressions within them.

Examples:

```
return x
```

Or:

```
return 1 + 1
```

1.2.2 Expressions

Primary Expressions

Catscript has eight kinds of primary expressions. These are broken into two categories.
Literals: Integer Literal, String Literal, Boolean Literal, List Literal, Null Literal,
Other: Identifier, Function Call, and Parenthesized Expressions.
Most of the primary expressions are literal expressions that correspond to the primitive types found in the Catscript type system.

Identifier Expressions are expressions that represent a keyword defined by the user:

x
foo
myVar

Integer Literal Expressions are expressions that represent integer numbers:

42
-144

String Literal Expressions are expressions that represent strings of characters:

"Hello World"
"This is a string in Catscript!"

Boolean Literal Expressions are expressions that represent the True and False symbols:

true
false

Expressions that signify a collection of Integer, String, Boolean, and List Literal Expressions are known as List Literal Expressions:

[1, 2, 3]
["Hello", "World"]

Null Literal Expressions are expressions that represent the null symbol. Null Literal Expressions are used when there is no value represented for a variable:

null

Expressions that contain information about what data to transmit to a function's parameters are called Function Call Expressions. Function Call Expressions signal the execution of a function during runtime.

foo(1, 2, 3)

Parenthesized Expressions are expressions that contain any type of expression inside two parentheses. The parentheses do not affect the contained expressions in any way:

("Hello" + "World")
(12 < 24)

Unary Expressions

Unary Expressions pertain to expressions that affect only one expression. The two symbols that can be used in a unary expression are the negative symbol and the not symbol (i.e., the '!' symbol), which can exclusively be applied to Integer Literals and Boolean Literals, respectively.

Examples:

-1

not True

Equality Expressions

Expressions that utilize a double equal or bang equal symbol to separate two expressions are referred to as Equality Expressions. The double equal symbol asserts that both sides are identical (have the same value), whereas the bang equal symbol asserts that both sides are distinct (have different values). These expressions may contain any expression type.

Examples:

True == True

True != False

Comparison Expressions

Expressions that use a less than, greater than, less than or equal to, or greater than or equal to symbol to separate two expressions are known as Comparison Expressions. These expressions only accept Integer Literals as the separated expressions.

Examples:

1 < 2

2 > 1

x <= y

y >= x

Additive Expressions

Expressions that use an addition or subtraction symbol (plus or minus, respectively) to separate two expressions are called Additive Expressions. These expressions accept Integer Literals, String Literals, or Parenthesized expressions that contain Integer or String Literals as the separated expressions. String Literals are exclusively capable of being added together, they cannot be subtracted. Examples:

"a" + "b"

12 - 11

Factor Expressions

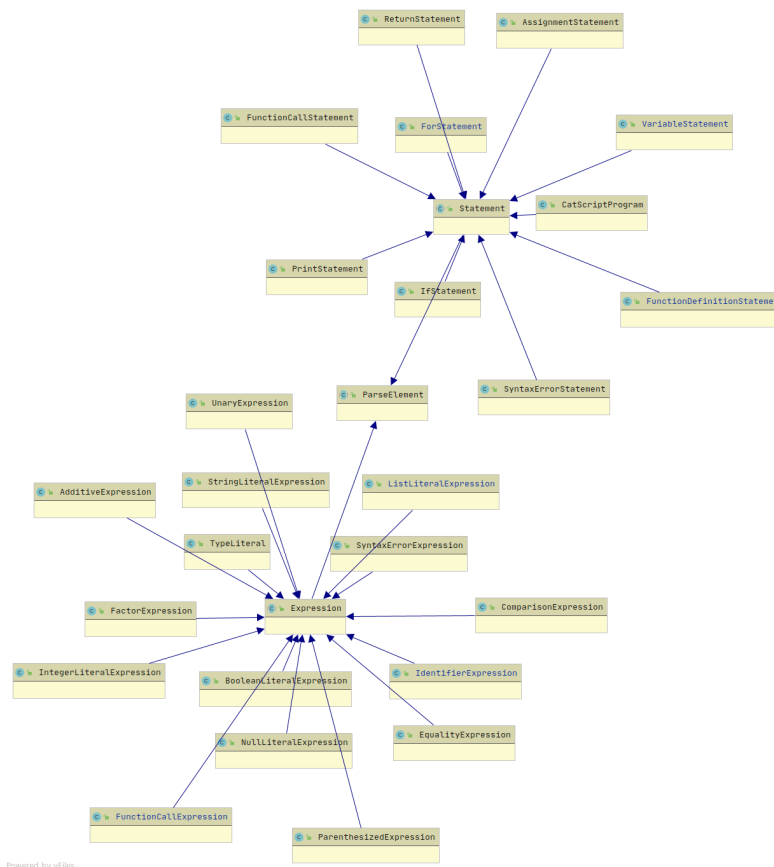
Factor Expressions refer to expressions that employ a multiplication or division symbol (asterisk or slash, respectively) to separate the two expressions. These expressions exclusively accept Integer Literals or Parenthesized Expressions that contain Integer Literals. Examples:

$8 * 2$

$15 / 3$

UML

The following UML diagram shows the overall structure of the Catscript Language. It shows that all of the expressions and statements extend the abstract classes expression and statement respectively. Additionally, both of the abstract classes extend the abstract class, parseElement, meaning that both expressions and statements are able to be parsed by the Catscript compiler.



Design Trade-offs

The primary decision point in this project's design was whether to implement a compiler utilizing recursive descent or parser generation techniques. Ultimately, the decision was made to use recursive descent, although either approach could have achieved the project's goal. Recursive descent was chosen because of the nature of the two methods.

With recursive descent, the programmer manually codes the various types of expressions and statements in a language and links them together to make the language coherent in its application and usage. In contrast, parser generation abstracts the concept of recursive descent by creating a parser to parse the rules of a language, creating a compiler with little effort required from the programmer. While parser generation is often a simpler method to create a compiler, it can be more difficult to understand and debug. Therefore, for this project, it made sense to implement a recursive descent compiler.

Software Development Life Cycle Model

Test driven development was the software development life cycle model utilized in this project. This model was well-suited to the project due to its well-defined steps. For each step, a set of tests was written, which followed along with the clear differences that appeared within the code itself. The code was divided into major sections, including tokenization, parsing, evaluation, and bytecode generation of the Catscript language. Moreover, the sections completed were dependent on their previous counterparts, which defined the development life cycle not only spatially but also temporally. By dividing the project into clear sections, the goals were defined more precisely, enabling an efficiently executed development cycle.