

CATSCRIPT COMPILER
CAPSTONE PROJECT REPORT

COMPILERS CSCI-468

Spring 2022

Rory McLean & Mason Medina
Gianforte School of Computing
Montana State University

Program

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

Teamwork

This project was broken up into five distinct sections of work. As this project used a test driven style of development, each section was accompanied by a set of tests that aligned to the work done during the associated time frame. The first four sections were completed, independently, by partner 1(Rory McLean), but on the fifth section, partner one and partner two(Mason Medina) collaborated by exchanging new tests that they had developed for any of the content created in the previous sections. In addition to the test trading, partner one and two collaborated on making the documentation for the project. Partner one created most of the documentation regarding the contents and structure of the Catscript Language, then partner 2 reviewed and made changes to it wherever relevant. Overall, for this project, the work is split between the two partners with partner one having done roughly 95% of the work and partner two having done roughly 5% of the work.

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 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 simple scripting language that is comparable to languages like Java and Python. The Catscript Compiler program is a program that takes in, processes, and executes code formatted in the Catscript language. The execution of that code is based on its Java interpretation and the compiler translates between them. An example of the Catscript language can be seen here with more examples shown in the features section:

```
var x = "Hello World"
print(x)
```

1.2 Features

1.2.1 Statements

For loops

For loops in Catscript are able to iterate over defined lists and perform operations per iteration. Operations are able to contain more types of statements such as print-statements and if-statements 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 are able to take and print its contents such as String or Integer Literals.

Example:

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

Output:

```
Hello World
```

Variable Statements

Variable Statements in Catscript are able to assign values to variable names. Any of the primary expressions are assignable to variable names. Types can be explicitly or implicitly defined when creating the variables.

Examples:

```
Var x = 1
```

Or:

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

Assignment statements

Assignment statements in Catscript are used when a variable is being assigned to a different value. Variables can only be re-assigned to the same type that they were implicitly or explicitly defined as, an Integer variable cannot be assigned to a String value.

Example:

```
Var x = 1
```

```
x = 3
```

Function Definition Statements

Function Definition Statements in Catscript are statements that define a function that is callable elsewhere in the program. The Function can be defined with parameters and must be defined with a body. The parameters can be defined explicitly or implicitly just like variable statements. The body can contain any number and types of statements. Functions are also able to contain return statements which should end the function. Functions with return statements should have a return type defined after the name and parameters of the function.

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

Return Statements in Catscript are statements that pass values from functions to where they were called in the greater scope. Return statements will end the function they are contained within. Return statements can have expressions contained within them.

Examples:

```
return x
```

Or:

```
return 1 + 1
```

1.2.2 Expressions

Primary Expressions

There are eight types of expressions defined under primary expressions: Identifier, Integer Literal, String Literal, Boolean Literal, List Literal, Null Literal, Function Call, and Parenthesized Expressions. Below, you will find short descriptions and examples of each.

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

```
x  
y  
z
```

Integer Literal Expressions are expressions that represent integer numbers:

```
42  
144
```

String Literal Expressions are expressions that represent strings of characters:

```
"Hello World"  
"I am alive"
```

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

```
True  
False
```

List Literal Expressions are expressions that represent a set of Integer, String, Boolean, and 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
```

Function Call Expressions are expressions that contain information about what information to send to an existing functions parameters. Function Call Expressions are used to 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 are expressions that are applied to only one expression. The two symbols that a unary expression can have, are the negative symbol and not symbol which can only be applied to Integer Literals and Boolean Literals respectively.

Examples:

```
-1  
not True
```

Equality Expressions

Equality Expressions are expressions that have a double equal or bang equal symbol separating two expressions, with the double equal symbol asserting both sides are the same and the bang equal symbol asserting both sides are different. The separated expressions may be any type of expression.

Examples:

```
True == True  
True != False
```

Comparison Expressions

Comparison Expressions are expressions that have a less than, greater than, less than or equal to, or greater than or equal to symbol separating two expressions. The separated expressions may only be Integer Literals.

Examples:

```
1 < 2  
2 > 1  
x <= y  
y >= x
```

Additive Expressions

Additive Expressions are expressions that have an addition or subtraction symbol, a plus or a minus respectively, separating two expressions. The separated expressions may be Integer Literals, String Literals, or Parenthesized expressions containing either Integer or String Literals. String Literals can only be added together and not subtracted.

Examples:

```
"a" + "b"  
2 - 1
```

Factor Expressions

Factor Expressions are expressions that have a multiplication or division symbol, an asterisk or a slash respectively, separating two expressions. The separated expressions may only be Integer Literals or Parenthesized Expressions containing Integer Literals.

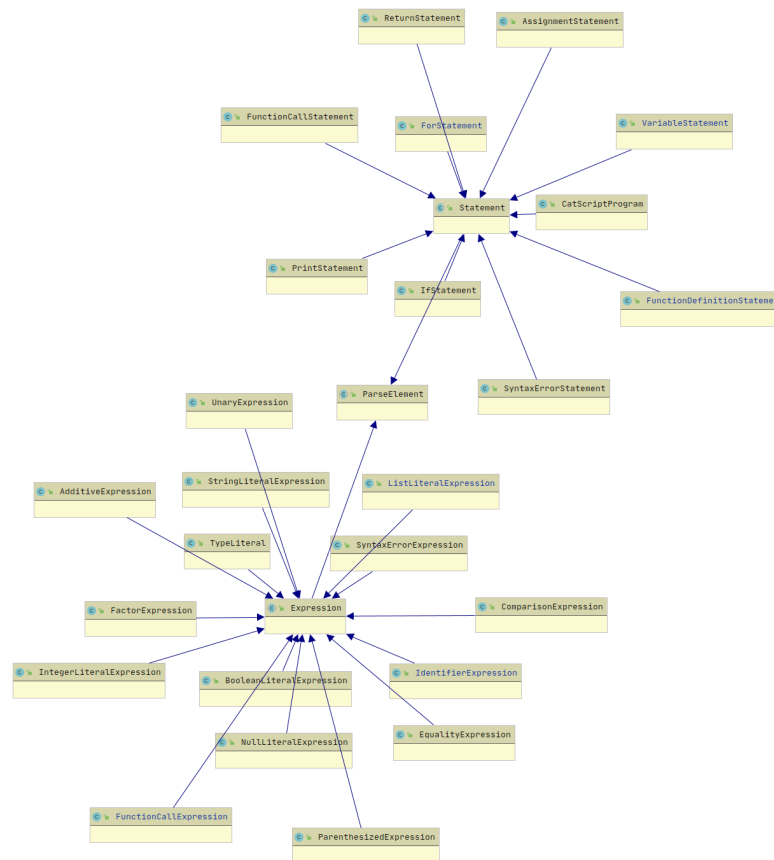
Examples:

5 * 6

30 / 5

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.



Powered by yFiles

Design Trade-offs

The main decision point in the design in this project regarded whether to implement a compiler that utilized recursive descent versus parser generation techniques. It was ultimately decided that recursive descent would be used over parser generation but either technique would have likely accomplished the goal of this project. The reason why recursive descent was used is due to the nature of both methods. The recursive descent method has the programmer code, by hand, the different types of expressions and statements of a language. The programmer then links the statements and expressions together so that the language is coherent in its applications and usages. Parser generation, however, abstracts the idea of recursive descent by creating a parser to parse the rules of a language, creating a compiler with very little effort from the programmer. Parser generation is often times a simpler, but more hands off way, to create a compiler that is harder to understand and debug. Because of these reasons, it made sense, for this class, to implement a recursive descent compiler.

Software Development Life Cycle Model

The software development life cycle model used in this project was test driven development. The process of this model worked very well for this project as its steps were very well defined. The set of tests that were written for each step of the project followed along with the clear differences that appeared within the code itself. The major sections of the code that was worked on were the tokenization, parsing, evaluation, and bytecode generation of the Catscript language. Additionally, the sections completed relied on their previous counterparts meaning that the development life cycle was not only defined spatially but also temporally. This distinction between sections of the project defined clearer and smaller goals allowing for an efficiently executed development cycle.