

Compilers, CSCI-468

Spring 2022 Semester

Alonso Darias, Miles Naberhaus

Section 1

A .zip file containing the “/src” directory of the code base for this project can be found at “/capstone/portfolio/source.zip” and the test file added by my partner can be found at “/test/java/edu/montana/csci/csci468/demo/PartnerTests.java”.

Section 2

Roles in development of this project were split between a developer (team member 1, Miles Naberhaus) and a tester (team member 2, Alonso Darias). Additionally, team member 2 provided technical documentation regarding the Catscript language. Development of the code base by team member 1 consumed the vast majority of total time spent on this project, with an estimated 100 hours spent. Team member 2’s additional tests and documentation consumed an estimated 6 hours of time. Using these estimates in time spent, contributions to this project are attributed to 94% member 1 and 6% member 2.

Section 3

A specific design pattern used inside of the code base can be found in the form of a memoization pattern implemented in the **CatscriptType.java** file, the relevant code is below.

```
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 new ListType(type);
}
```

The purpose of using memoization is in avoiding computing answers multiple times, such that they are stored somewhere accessible when we need the computation again. Here, we are using memoization to avoid having to check the typing of a list repeatedly, by instead storing the relevant value in a cache when we first compute a new solution. This cache is referred to on subsequent calls in order to avoid performing this action again.

Section 4 (documentation on following pages)

Catscript Guide

This document should be used to create a guide for catscript, to satisfy capstone requirement 4

Introduction

Catscript is a simple scripting language built on top of the Java programming language. Here is an example of a simple Catscript program:

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

Features

Catscript is made up of expressions and statements:

Expressions

Boolean Literal Expression: A boolean literal expression is used to represent a boolean value.

Example code:

```
// Assigning a boolean value to a variable  
var booleanVariable = true
```

Integer Literal Expression:

An integer literal expression is used to represent an integer value.

Example code:

```
// Assigning the integer value '1' to a variable  
var integerVariable = 1
```

List Literal Expression:

A list literal expression is used to represent a list value. A list literal can have an explicit type declaration or be initialized with an implied type.

Example code:

```
// List with implicit type
[1, 2, 3]
// Assigning a variable a list value with an explicit type
var listVariable : list<int> = [1, 2, 3]
```

Null Literal Expression:

A null literal expression is used to represent a null value.

Example code:

```
// Assigning a null value to a variable
var nullVariable = null
```

String Literal Expression:

A string literal expression is used to represent a string value.

Example code:

```
// Assigning a string value to a variable
var stringVariable = "This is a string"
```

Type Literal Expression:

A type literal expression represents one of the six basic Catscript types: *INT*, *STRING*, *BOOLEAN*, *OBJECT*, *VOID*, and *NULL*. It used to declare an explicit type.

Example code:

```
// Assigning an INT value to a variable with an explicit type literal
var integerVariable : int = 1
```

Additive Expression:

An additive expression is used to add integer values or concatenate strings as it is in Java. It consists of a left-hand value, a right-hand value, and an operator: either plus (+) or minus (-).

Example code:

```
// Adding two integers. This will output "2"  
print(1 + 1)  
// Concatenating two strings. This will output "Hello, World"  
print("Hello, " + "World")
```

Factor Expression:

A factor expression is used to multiply or divide integer values. It consists of a left-hand value, a right-hand value, and an operator: either multiply (*) or divide (/).

Example code:

```
// Multiplying two integers. Outputs 2  
print(1 * 2)  
// Dividing two integers. Outputs 1  
print(2 / 2)
```

Parenthesized Expression:

A parenthesized expression is any simple or compound expression surrounded by parentheses. This allows the user to manipulate the precedence of operators.

Example code:

```
// A parenthesized expression. Outputs 12  
print((2 * (2 + 4)))
```

Comparison Expression:

A comparison expression evaluates a boolean value according to the comparison between two values. A comparison expression can use either, greater than (>), less than (<), greater or equal to (>=), or less or equal to (<=).

Example code:

```
// A comparison expression. Outputs true  
print(1 < 2)
```

Equality Expression:

An equality expression evaluates a boolean value according to the equality of two values. It can use one of two operators: equals (==) or not equals (!=).

Example code:

```
// An equality expression. Outputs true
print(true == true)
```

Unary Expression:

A unary expression is an expression with one value. It can be either a logical or numerical negation.

Example code:

```
// A numerical unary expression. Outputs -1
print(-1)
// A logical unary expression. Outputs false
print(not true)
```

Identifier Expression:

An identifier expression is a string which represents either a function or a variable.

Example code:

```
// A function identifier with the string value "function"
function()
// A variable identifier with the string value "variable"
var variable = 1
```

Function Call Expression:

A function call expression is an expression that contains the identifier of a previously defined function with the arguments required for that function in parentheses.

Example code:

```
// A function call expression representing a function called "add" which takes two arguments
add(2, 2)
```

Syntax Error Expression:

A syntax error expression is an expression that represents something a user enters which cannot be parsed into any other expression.

Example code:

```
// A program line that would parse into a syntax error expression because it is un terminated list  
var list = [1, 2, 3,
```

Statements

Return Statement

A return statement is used in a function definition statement to return a value to the statement which called the function.

Example code:

```
// A return statement which a null value  
return null
```

Function Definition Statement

A function definition statement is used in define what is to be executed when a function call statement is executed. It consists of the keyword "function", an identifier for the function, a list of required arguments, a list of statements to be executed in the function, and a return statement.

Example code:

```
// A simple function called printBool which takes a boolean argument, prints, and returns that value  
function printBool(value : bool) {  
    print(value)  
    return value  
}
```

Function Call Statement

A function call statement is used to execute a function that has been defined with a function definition statement. It consists of the function identifier expression, followed by the set of required arguments for that function.

Example code:

```
// A function call statement referencing the function printBool  
printBool(true)
```

For Statement

A for statement is used for iterating over a set of values while executing a set of statements for each value in the set.

Example code:

```
// A for statement which will print out the numbers 1 through 3 in order  
for (i in [1, 2, 3]) {  
    print(i)  
}
```

If Statement

An if statement is used to conditionally execute a set of statements based on the truth of any expression that returns a boolean value. It can optionally have an else statement which will execute when the boolean condition is false. Any else statement can also contain an if statement in itself.

Example code:

```
// An if statement that will print the value "Hello, world" when booleanVariable has  
// the value true, and "Goodbye, world" when booleanVariable has the value false  
if (booleanVariable) {  
    print("Hello, world")  
} else {  
    print("Goodbye, world")  
}
```

Print Statement

A print statement is used to print a string to the output. It may contain a literal or an expression which will evaluate to a literal.

Example code:

```
// A print statement which outputs the value "Hello, world"
var stringValue = "Hello, world"
print(stringValue)
```

Variable Statement

A variable statement is used to define a variable. It consists of the keyword "var", followed by an identifier for the variable, the assignment operator (=), followed by an expression which will be evaluated and assigned to the variable. Variable statements can either use an explicit type or have the type implied from the value of the expression.

Example code:

```
// A variable statement which assigns an integer value with an implicit type
var numberValue = 1
// A variable statement which assigns an integer value with an explicit type
var integerValue : integer = 1
```

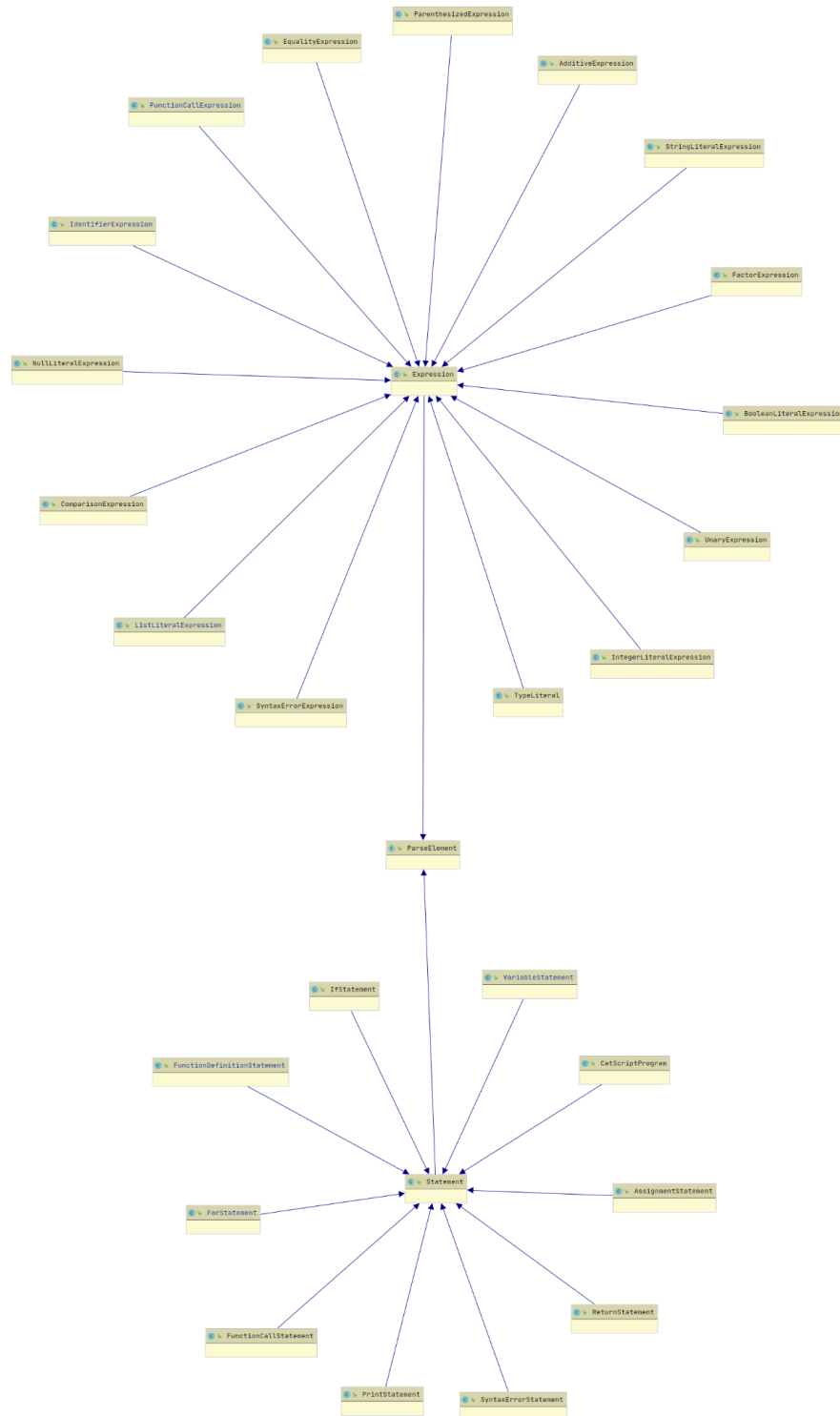
Assignment Statement

An assignment statement is used to assign a value to a variable which has already been defined. It consists of an identifier for a variable which has been defined previously, followed by the assignment operator (=), and an expression which will be evaluated and assigned to the variable. The new value must be of a type that is consistent with the type that the variable was assigned at definition.

Example code:

```
// An assignment statement to give the variable numberValue the value of 2
numberValue = 2
```

Section 5



Powered by yfiles

The above UML diagram details the class structure of parse elements in Catscript. Both the [Expression](#) and [Statement](#) classes are extended from [ParseElement](#), which forms the basic manner in which Catscript elements are parsed. From these implementations are all of the specific classes for each class of statement and expression, for instance [ComparisonExpression](#)

and `EqualityExpression` are both extended from `Expression`, while `IfStatement` and `ForStatement` are both extended from `Statement`. As these elements are parsed, the specific aspects of their implementations are evaluated with regard to their individual needs. Those elements that fall into the Expression category have their own unique implementations of an evaluation process. For instance, most literal expressions, such as `BooleanLiteralExpression` and `StringLiteralExpression` essentially have an evaluation process which returns what value they contain (a boolean value or a string respectively). On the other hand, logical expressions such as `ComparisonExpression` and `EqualityExpression` have an evaluation process by which they must compare values and return a boolean based upon this evaluation. Those elements that fall into the class of statements, such as a `ForStatement`, instead have an implemented execution, by which they will execute some manner of computation. In this case, a `ForStatement` will iterate over a body of statements by an iterator determined by some expression, executing each of these statements during each iteration.

Section 6

A major design decision made regarding this project was the decision to implement the recursive descent parsing algorithm rather than a parser generator, which is more often taught in college courses. This decision was made with several things in mind, with a major favorable aspect of recursive descent being the ease of understanding it requires in comparison to a parser generator. Recursive descent also has parse trees that are much easier to use in debugging code, and tracing through the tokenizing and parsing process is readable and simple. While these are huge advantages of recursive descent for our purposes, one comparative advantage of a parser generator implementation is its smaller size. With less code needed for a parser generator compared to recursive descent, the overall development would be overall not as large, though potentially more complex. However, much of this was mitigated by being provided much of the basic foundation needed to create the recursive descent algorithm. With a foundation to build upon being provided, the simplicity of recursive descent provided major advantages in finalizing its functionality through the tokenizing and parsing processes.

Section 7

Much like other classes I have taken with Carson Gross, development of the final project was driven by test based development. Essentially, there are several files with similar unit tests acting as checkpoints in order to drive the programming and development process. Using these tests as a template makes development much more straightforward and provides a simple goal state to reach. Other than simply making programming more efficient, it enables students to ask much better questions and receive more directly applicable answers than they might be able to if they were asking without as clear of a goal in mind. Even without asking these questions, the use of these test bases gives a clear direction in mind to what the code in question should be doing. For example, if the test specifies that we are asserting the value of evaluating an expression to some return value, it allows a developer to template out code with that return value in mind. If

the value needs to be a string, it narrows down the development for some function from being potentially vague to a much more direct practice of some operations with a string return.

These types of tests also provide an immediate example of input/output operations that a program needs to accomplish. In my experience with Gross' class projects (for which I have had a systems project, a database project, and finally a compilers project), the addition of these examples has always provided a colossal benefit towards understanding how to construct the code base. When a test fails, the feedback from the error allows for the addition of far more accurate breakpoints to simplify and speed up the debugging process. Without using this test-driven approach to development, my current ability to use debugging tools would be nowhere near as capable as it is now.

Another aspect of the manner in which this test driven development process is extremely helpful is the addition of heavy use of the version management tool github. Through these classes, the use of github to first create a cloned github from the class master, receive grades through this clone, and upload code privately are all important tools in modern development I feel much more confident using. Seeing the tests built and evaluated both locally and in the repository made for a greater deal of knowledge regarding github automation as well. All of these tools have been crucial to increasing my capacity as a competent programmer.