

CSCI-468 Capstone Report

Section 1: Program

The compiler is contained in source.zip, which is in the GitHub repository in the portfolio folder with this report.

Section 2: Teamwork

For this project we were to collaborate with a teammate to do the following:

- Create a small suite of tests for the Catscript Compiler.
- Write a technical summary of the Catscript language.

My partner and I each wrote three tests for each other involving 2D Arrays. The tests are included in the tests section in the “partners” package.

Section 3: Design Pattern

Within this project we used memorization, otherwise known as the Flywheel pattern. As a brief refresher, memorization is a process of optimization where repetitive operations are able to be skipped over. In the case of our Catscript Compiler, we created a cache, in which we want to store all the list types being used in our scrip9t. Rather than always add a list type to the cache upon creation, we memorize this call to first check if the type is in the cache already, and if it is not, then add it, otherwise, do nothing. This is an example of memorization.

Section 4: Technical Writing

The following is a transcript from Technical Writup.md.

```
# Catscript Guide
```

```
This document is a guide for Catscript, to satisfy capstone requirement 4
```

```
## Introduction
```

```
The Catscript language is a very simple programming language designed around the use of recursion within the compiler. The language supports the use of for loop statements, if/if-else statements, print statements,
```

assignment statements, and function statements. Catscript also only supports the following types: int, string, bool, object, and list<type>.

Features

Below are details of how each part of the compiler functions.

For Statement

```
for (x in list) { print(x) }
```

The For Statement begins with the token 'for' followed by parenthesis containing an identifier, the token 'in', and an iterable value. The body of the for loop contains statements to be executed and is surrounded by brackets. First, the loop evaluates the expression to be iterated over (list in the example). On every iteration, the value of x is set to the current index of the expression. Finally, every statement in the body of the for loop is executed.

If Statement

```
if (x > 0) {  
    print("positive")  
} else if (x == 0) {  
    print("zero")  
} else {  
    print("negative")  
}
```

The If Statement begins with the 'if' token, followed by parenthesis containing an expression. The body of the if loop, surrounded by brackets, contains statements to be executed if the expression evaluates to be true. Following the close bracket, the token 'else' can be used to execute statements if the expression is false, or another if statement can be created if the 'else' token is followed by the 'if' token.

Print Statement

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

The Print Statement begins with the token 'print' followed by parenthesis containing the expression to print. It works by first evaluating the expression inside the parenthesis and then prints the result to the program.

Variable Statement

```
var x = 10  
var x : int = 20
```

The Variable Statement begins with the 'var' token followed by an identifier. If the type of the variable is to be declared, the 'colon' token and the type follow the identifier. Next for both cases is the 'equal' token ending with the expression. The variable statement works by

evaluating the expression that is going to be assigned to the variable, then that expression's value is set to the variable. If the type is not specifically declared (as in the second example), the type is inferred from the type of the expression.

Function Call Statement

The Function Call Statement is the statement that is calling the function.

Assignment Statement

```
```
```

```
x = null
```

```
```
```

The Assignment Statement begins with an identifier, followed by the 'equal' token, ending with an expression. The expression is evaluated and the value set to the identifier.

Function Declaration

```
```
```

```
function foo(a, b, c) : int {
 result = a + b + c
 return result
}
```

```
```
```

The Function Declaration defines a function that can be called later in the code. It begins with the 'function' token, followed by an identifier, parenthesis with an argument list inside, an optional 'colon' token and type definition, followed by brackets that contain the function body statements.

Function Body Statement

```
```
```

```
result = a + b + c
```

```
```
```

```
```
```

```
return result
```

```
```
```

The Function Body Statement is the body of a function. It can either be a statement like the first example given or it can be a return statement like in the second example.

Return Statement

```
```
```

```
return x
```

```
```
```

The Return Statement begins with the token 'return' followed by an expression. If a return statement is reached, the expression is evaluated and returned.

Expressions

Expressions are designed recursively, starting at the equality expression and recursively calling the next expression down in the grammar until a primary expression is found, then return that value to the top.

Equality -> Comparison -> Additive -> Factor -> Unary -> Primary

Equality Expression

```
xxx
x != 10
y == 15
xxx
```

The Equality Expression is some expression followed by either the 'equal_equal' or 'bang_equal' token, followed by another expression. These two expressions are evaluated and compared to see if they are equal or to see if they are not equal.

Comparison Expression

```
xxx
x > 10
y < 10
a >= 10
b <= 10
xxx
```

The Comparison Expression starts with an expression followed by either the 'less', 'less_equal', 'greater', or 'greater_equal' tokens ending with another expression. These two expressions are evaluated and compared to see if the first expression is less than/greater than/less or equal to/greater or equal to the second expression.

Additive Expression

```
xxx
x + y
1 - 5
xxx
```

The Additive Expression starts with an expression followed by either the 'plus' or 'minus' tokens ending with another expression. The two expressions are evaluated and then added together or the second expression is subtracted from the first.

Factor Expression

```
xxx
a * b
10 / 2
xxx
```

The Factor Expression starts with an expression followed by either the 'star' or 'slash' token ending with a second expression. The expressions are evaluated and either multiplied together or the first expression is divided by the second expression.

Unary Expression

```
xxx
not true
-10
xxx
```

The Unary Expression Starts with either the 'not' or the 'minus' token followed by some expression. If the token 'not' is used, the expression value switches from true to false and false to true. If the 'minus' token is used, the expression value is negated.

Primary Expression
```

x  
"string"  
10  
true  
```

The Primary Expression is one of the following tokens: 'identifier', 'string', 'integer', 'true', 'false', 'null', or it can be a list literal, function call, or an expression surrounded with parenthesis.

List Literal
```

[x, y, z]  
[1, 2, 3]  
["these", "are", "strings"]  
```

The list literal is a list of expressions contained within braces separated by commas. Each value of the list is evaluated to assign type values to it.

Function Call
```

functionName(x)  
functionName1(x, y, z)  
```

The Function Call begins with the function name, an identifier, followed by parenthesis containing the argument or the list of arguments. This invokes the arguments on the function being called and returns the result.

Argument List
```

a, b, c  
```

The Argument List is a list of expressions separated by commas that are used as arguments in a function call.

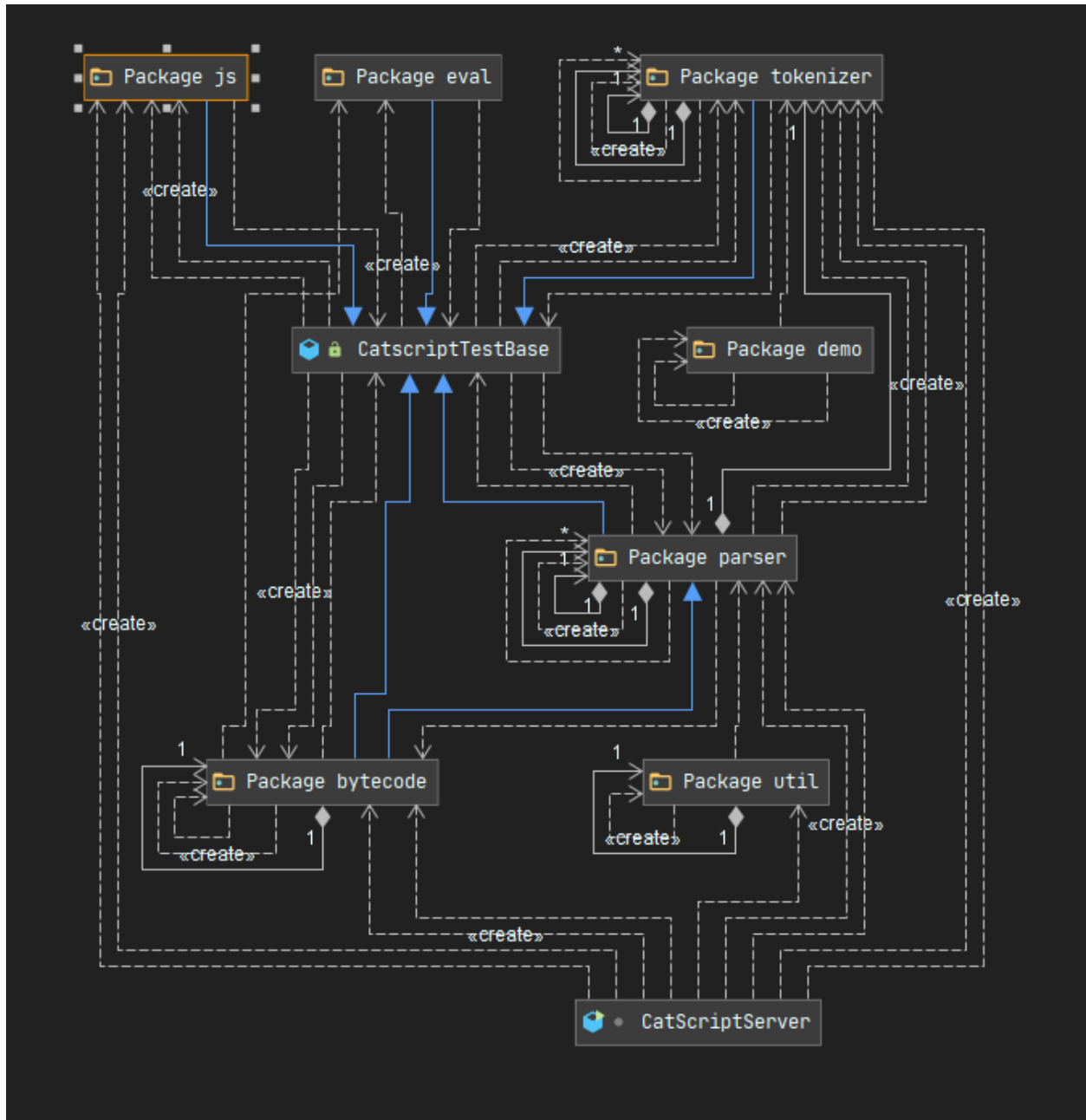
Type Expression
```

int  
string  
object  
list<int>  
```

The Type Expression is how the value's type is defined in Catscript. The types available are as follows: 'int', 'string', 'bool', 'object', and 'list<type_expression>'. A list type has the addition type definition for the values in the list.

Section 5: UML

Below is the UML of the parse elements as prescribed.



Section 6: Design Trade-offs

While we used a recursive descent parser for Catscript, typically, compilers built in schools are built using what is called a parser generator. Parser generators are typically chosen because they are technically a more efficient way of parsing, however, recursive descent parsers are typically more intuitive and universal. So, in using a recursive descent parser for Catscript, we are trading efficiency for universality and ease of use.

Section 7: Software Development Lifecycle

For this project, we used a test-driven design lifecycle. For each module, we were given a suite of tests which tested the important aspects and functionalities of each module. The module was considered complete when all tests were passing. This method helped our team stay on track.