

CSCI 468 - Compilers
Spring 2022
Prof. Carson Gross
Jacob Conelly, Wyatt Wright

Section 1: Program

A zip file of the final repository is included in this directory.

<https://github.com/wyattrig/csci-468-spring2022-private/blob/master/capstone/portfolio/source.zip>

Section 2: Teamwork

The teamwork section of this project required each member to create documentation for the catscript programming language. This language was created for the compilers class where the goal was to build a compiler that could compile catscript down to Java byte code. Each partner submits the other person's documentation, so my partner's documentation is in Section 4 of this document. Additionally, we were required to generate a test suite of 3 Java tests that evaluated some part of the compiler's ability to work with catscript. My partner chose to test that comparison, additive, and factor expressions worked together, functions could be called inside functions, and booleans within for statements evaluated properly. The first test I submitted for my partner tested his compiler's handling of order of operations, negative integers, and the rounding up or down of fractional division results. The next two tests were examining variable scope within functions and checking that conditional statements behave correctly when chained. I'd estimate that we both spent about 5 hours writing tests and putting together documentation, and the effort was split equally both ways since we were exchanging work. Additionally, in the past we had both helped each other debug our code.

Section 3: Design pattern

The design pattern we used in this project is called flyweight and it optimizes the getListType method within the catscript type class. This pattern is a type of memoization and its goal is to reduce expensive function calls and limit object generation, saving time and memory. This is accomplished by implementing a simple caching system where, when a list is created in a program, instead of simply creating a new list type object for each new list, first we check our cache to see if a prior list has been instantiated with the same type. If it has, we reuse that list type object for the new list.

```
// Memoized Call
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;
}
```

CatScript Documentation

The CatScript Type System

CatScript is a strongly typed language. Meaning once a variable is declared the variable retains its type. For example, if you declare an int you cannot assign that variable to a Boolean type of true.

CatScript Types:

int – a 32 bit integer

string – a java-style string

bool – a boolean value

list – a list of values with a type 'x'

null – the null type

object – any type of value

Variables can be declared by inferring types or explicitly declaring the type outlined below. When a variable is declared without an explicit type it will be inferred based on the data type found on the right-hand side.

```

1 // Declaring an int
2 var a = 10
3 var b: int = 10
4
5 // Declaring a boolean
6 var c = true
7 var d: bool = false
8
9 // Declaring a string
10 var e = "string"
11 var f: string = "string"
12
13 // Declaring a null value
14 var g = null
15 var h: int = null
16
17 // Declaring an object
18 var i: object = "string"
19 var j: object = 10

```

Note: A variable of type object must be explicitly declared otherwise CatScript will infer the type from the right-hand side.

CatScript List Type:

The CatScript list type can be declared in multiple ways. You can declare a list with an implicit type or with an explicit type. It is also possible to create a list of objects that does not need to contain the same type. A list declared as an explicit type must be composed solely of that type. For example, if a list is declared as type int it cannot contain a boolean value.

```

1 // Declaring list with implicit type
2 var a = [1, 2, 3]
3 var b = ["string1", "string2", "string 3"]
4
5 // Declaring list with explicit type
6 var c: list<int> = [1, 2, 3]
7 var d: list<bool> = [true, false, false]
8
9 // Declaring list of objects
10 var e: list<object> = ["string", 1, true]

```

CatScript Boolean Operators

Operator	Description
== (equal to)	Checks if the value of two operands are equal and returns true if equal and false if not equal.

<code>!=</code> (not equal to)	Checks if the value of two operands are not equal and returns true if they are not equal or false if they are equal.
<code>></code> (greater than)	Checks if value on the left-hand side is greater than the value of the right-hand side, if yes it will evaluate to true.
<code>>=</code> (greater than or equal to)	Checks if value on the left-hand side is greater than or equal to the value on the right-hand side. If yes return true.
<code><</code> (less than)	Checks if value on the left-hand side is less than the value on the right-hand side. If yes return true, otherwise return false.
<code><=</code> (less than or equal to)	Checks if the value on the left-hand side is less than or equal to the value on the right-hand side. If yes return true.

Example:

```
1 var x = 5
2 var y = 0
3
4 // Equal to.
5 print(x == y) // false
6
7 // Not equal to.
8 print(x != y) // true
9
10 // Greater than.
11 print(x > y) // true
12
13 // Greater than or equal to.
14 print(x >= y) // true
15
16 // Less than.
17 print(x < y) // false
18
19 // Less than or equal to.
20 print(x <= y) // false
```

CatScript Negation:

The CatScript “not” expression is used to negate a boolean value. For example, if var x is set to true and we negate x with the “not” expression, then x would become false.

Example:

```
1 // Example of negating during an assignment statement.
2 var x = not true // false
3
4 // Example using a variable in an if statement.
5 var y = false
6
7 if (not y) {
8
9     print(y) // false
10
11 }
```

CatScript If-Else Logic

Like most other languages CatScript can perform Boolean logic using If-Else statements. CatScript does not support an Else-If statement. Instead, an If-Else statement can be imbedded in another If-Else statement.

If Statement:

Use the If statement to specify a block of code to be executed if the condition is true.

Example:

```
2 if(condition) {
3
4     // block of code to execute
5
6 }
```

```
2- if(3 > 1) {  
3  
4     print("3 is greater than 1")  
5  
6 }
```

Else Statement:

Use the Else statement to specify a block of code to be executed if the condition is false.

```
2- if(condition) {  
3  
4     // block of code to be executed if true  
5  
6 }  
7- else {  
8  
9     // block of code to be executed if false  
10  
11 }
```

Example:

```
1 var hour = 9  
2- if(hour > 12) {  
3  
4     print("Good Afternoon!")  
5  
6 }  
7- else {  
8  
9     print("Good Morning!")  
10  
11 }
```

CatScript Functions

A CatScript function is a block of code that runs when it is called. Data called parameters can be passed into the method. Methods can also return a value to be used where the function was called.

Functions consist of five parts the keyword **function**, the function name, zero to many parameters, a return type or void for no return, and the function body. Refer to the function outline below.

```
1
2 function functionName(parameters): returnType {
3
4     // Code to execute.
5
6 }
```

Example:

```
2 // Function with void return that prints a concatenated string.
3 function foo(str1: string, str2: string) {
4
5     print(str1 + str2)
6
7 }
8
9 // Function with int return type that adds two int paramaters.
10 function add(a: int, b: int): int {
11
12     return a + b
13 }
14
15 foo("foo", "bar") // foobar
16 var x = foo(1, 1)
17 print(x) // 2
```

The function foo is declared with two parameters of type string and called with two strings passed in on line 15. When foo is called it concatenates the strings and prints the result. It has a return type of void which is denoted by not adding a return type like in the add function.

CatScript For Loop

The for loop is used to iterate/traverse over lists. CatScript allows you to iterate over a list declared in the for loop or over a variable referencing a list.

```
2 for(var in list) {  
3  
4     // code to execute using var  
5  
6 }
```

Example:

```
2 for(x in [1, 2, 3]) {  
3  
4     print(x)  
5  
6 }  
7  
8 var array = [1, 2, 3]  
9 for(x in array) {  
10  
11     print(x)  
12  
13 }
```

Both these for loops print out “1 2 3”. The loop will iterate over the list and execute the code within loop and when complete move onto the next index in the list.

Additive and Factor Expressions

CatScript supports the addition (+), subtraction (-), multiplicative (*), and division (/) operators.

Note: The addition operator is also overloaded to support string concatenation.

Example:

```
1 // Additive operator.
2 var a = 1 + 1
3
4 // Subtraction operator.
5 var b = 1 - 1
6
7 // Multiplication operator.
8 var c = 1 * 1
9
10 // Division operator.
11 var d = 1 / 1
```

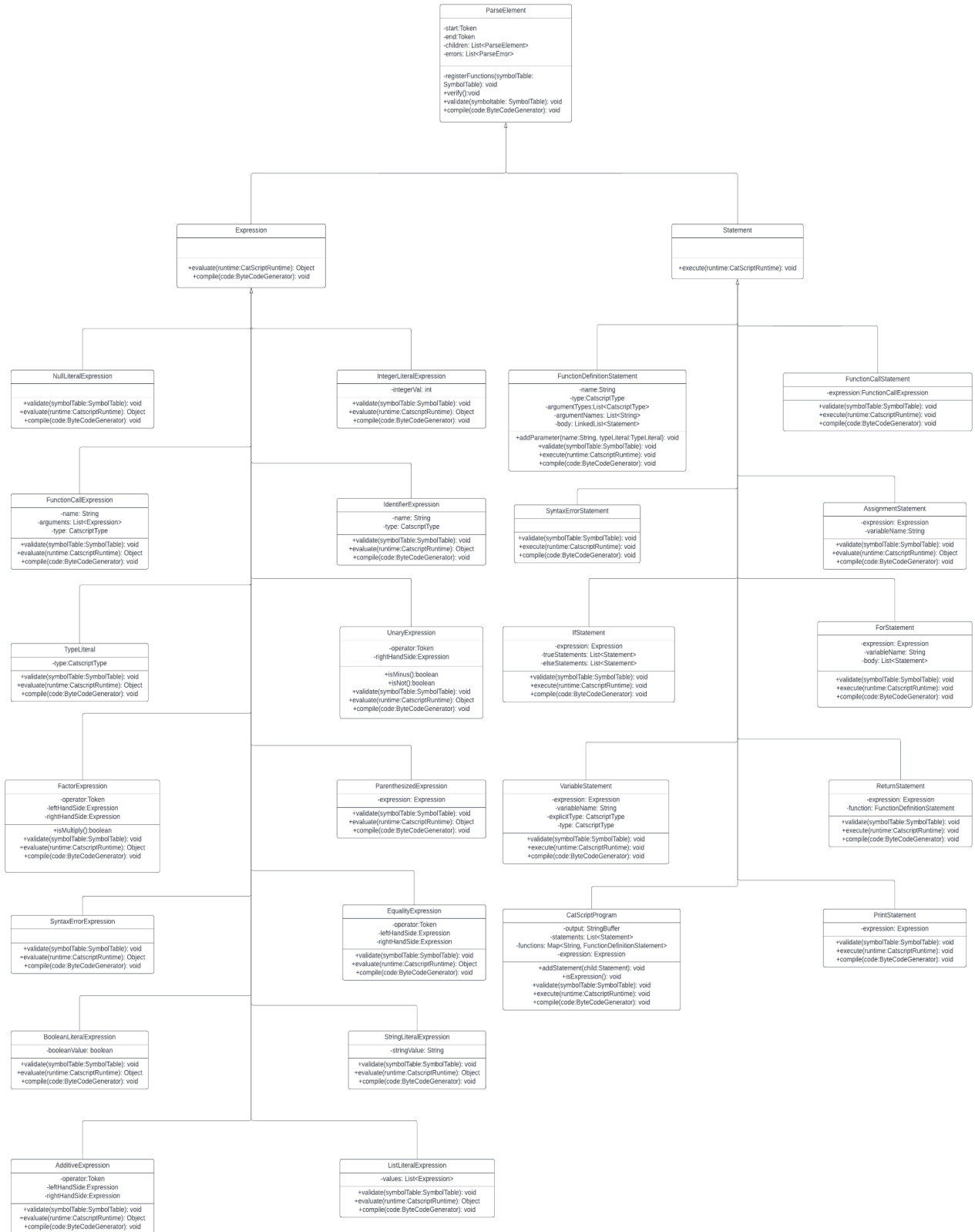
String Concatenation

Like many other languages CatScript overloads the plus (+) operator to perform string concatenation. To concatenate string simply put a plus (+) operator between the two values you wish to concatenate.

Note: Concatenation will still work if a string is concatenated with a boolean, null, or int value.

```
1 // String concatenation with another string
2 var a = "foo" + "bar" // "foobar"
3
4 // String concatenation with int
5 var b = "foo" + 1 // "foo1"
6
7 // String concatenation with boolean
8 var c = "foo" + true // "footrue"
9
10 // String concatenation with null value
11 var d = "foo" + null //foonull
```

Section 5: UML.



This UML class diagram shows the structure of the parse elements. Each parse element is created by the Catscript parser based on tokens read in by the tokenizer. Mirroring the grammar, there are two sections of parse elements that inherit from either the statement or expression classes. These elements are the implementations of the individual statements or expressions and they implement different methods depending on their type. The most common methods for the statements are validate, execute and compile. The most common methods for the expressions are validate, evaluate, and compile. In order to keep the uml simple and readable I chose not to include getters and setters in the class functions. I also did not show associations to the catscript type or the parent expression or statement classes for the same reason. Additionally LucidChart restricts the amount of associations that can be placed with a paywall. Most classes behave similarly so I'll only walk through one example. The function definition statement has a type variable that references the catscript type class which would be a 1 to 1 association. It also has a list of catscript types called arguments which would be a 1 to many association with the catscript type class and a list of statements which would also be a 1 to many association with the statement class.

Section 6: Design trade-offs

One of the major design tradeoffs for this project design was the overall method of implementation for the compiler. The method we used in this class is called recursive descent. This algorithm takes advantage of the recursive nature of language grammars and makes the process of creating parse trees intuitive. The alternative is generative parsing which is more commonly taught. A generative parser is much less hands on since this method uses tools that take a language specification and generates the different parts of a compiler like a lexer and a parser. Choosing this approach would have required less infrastructure, but it would have resulted in less hands-on work coding the compiler and less intuition about what's actually happening under the hood. Additionally, while generative parsing is common in an academic setting, recursive descent parsing is more widely used in-industry.

Section 7: Software development life cycle model

For this project we used test driven development. The idea with test driven development is to write tests before creating any actual code. If you're implementing a piece of program functionality, you would begin by writing a test asserting the expected output of that functionality. Then you would write the code and get the test to pass. I personally liked using test driven development for a few different reasons. The first reason was that it was explicitly clear what was expected from the program. The second reason was that it is easy to maintain motivation with this development model since the test suite gives a visual indication of development progress. Running a test and having everything pass is very satisfying. Additionally, having the tests in place is a major help for debugging since it allows a developer to see if any new changes break a piece of functionality that was previously working. One of the big downsides with test driven development is that the quality of the program is based largely on the quality of the tests. It's possible to implement a solution that will pass the test but not result in the correct

behavior all the time or cause issues down the road when implementing other parts of the program.