

# **CSCI 468: Compilers**

**Spring 2022**

William Popp, Emilia Bourgeois

## Section 1.

Source file: **"\capstone\portfolio\source.zip"**

## Section 2.

Team member 2 contributed to the project by providing 3 written java tests using the J-Unit testing library. These tests were designed to check for basic functionality of our Catscript compiler. The tests were successfully run at the completion stage of our project and can be found at **"\src\test\java\edu\montana\csci\csci468\demo\PartnerTests.java"**. Team member 2 also filled out and completed the documentation for our compiler. Providing detail for using the language was important for any user to interact and develop in Catscript. Team member 1 contributed to the design of the tokenizer, parser, evaluation, and bytecode. These four elements were the bulk of the project and are what make up Catscript. Team member 1 also contributed to the UML diagram for Catscript. In total team member 1 averaged 90% of the time spent on the project while team member 2 averaged 10% writing documentation and tests for Catscript.

## Section 3.

In the Catscript compiler, we used the memoization pattern. This pattern was used specifically for list types because the types of lists are not stored, so we created a map to store these types and access them later. I chose to make this method and hashmap thread safe for good practice by using concurrent hashmaps and the "putIfAbsent" method. The file can be accessed under **"\src\main\java\edu\montana\csci\csci468\parser\CatscriptType.java"**. I have attached the file below where the memoization pattern is highlighted in yellow.

```

package edu.montana.csci.csci468.parser;

import java.util.List;
import java.util.Objects;
import java.util.concurrent.ConcurrentHashMap;

public class CatscriptType {

    public static final CatscriptType INT = new CatscriptType("int", Integer.class);
    public static final CatscriptType STRING = new CatscriptType("string", String.class);
    public static final CatscriptType BOOLEAN = new CatscriptType("bool", Boolean.class);
    public static final CatscriptType OBJECT = new CatscriptType("object", Object.class);
    public static final CatscriptType NULL = new CatscriptType("null", Object.class);
    public static final CatscriptType VOID = new CatscriptType("void", Object.class);

    private final String name;
    private final Class javaClass;

    public CatscriptType(String name, Class javaClass) {
        this.name = name;
        this.javaClass = javaClass;
    }

    public boolean isAssignableFrom(CatscriptType type) {
        if (type == VOID) {
            return false;
        } else if (type == NULL) {
            return true;
        } else if (this.javaClass.isAssignableFrom(type.javaClass)) {
            return true;
        } return false;
    }

    // TODO memoize this call
    static ConcurrentHashMap<CatscriptType, ListType> cache = new ConcurrentHashMap<>();

    public static CatscriptType getListType(CatscriptType type) {
        return cache.computeIfAbsent(type, catscriptType -> new ListType(type));
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        CatscriptType that = (CatscriptType) o;
        return Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    public Class getJavaType() {
        return javaClass;
    }

    public static class ListType extends CatscriptType {
        private final CatscriptType componentType;
        public ListType(CatscriptType componentType) {
            super("list<" + componentType.toString() + ">", List.class);
            this.componentType = componentType;
        }

        @Override
        public boolean isAssignableFrom(CatscriptType type) {

```

# Section 4: Technical Writing

---

## Catscript Guide

---

This document is used as a guide for catscript, to satisfy capstone requirement 4.

### Introduction

---

Catscript is a simple scripting language. Here is an example:

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

### Features

---

#### Types

Catscript has five types and one complex type: - int - a 32 bit integer - string - a java-style string - bool - a boolean value - list - a list of value with the type 'x' - null - the null type - object - any type of value

Variables can be declared with and without types

```
var x = "foo"
var x : string = "foo"
```

Lists are special types which also store a type of the list which can be any of the other five. Declared using:

```
var x : list<Type> = [1,2,3]
```

#### Basic Expressions

- + addition
- - subtraction
- \* multiplication
- \ division
- not not
- - negate
- true
- false
- null

Examples:

```
var q = -1 + 1 * 4
if(not x)
if(x!=null)
else if(false)
```

#### If & Else statements

If statements can be used alone like so:

```
if(x){
    var y = 10
}
```

With an else:

```
if(x){
    var y = 10
}
else {
    var y = 20
}
```

Or with an else if:

```
if(x){
    var y = 10
}
else if(z){
    var y = 20
}
```

and all can be mixed and matched as long as an if statement is first and if there is an else it is only at the end like so:

```
if(x){
    var y = 10
}
else if(z){
    var y = 20
}
else {
    var y = 30
}
```

## Equality & Comparison Expressions

Catscript supports basic equality and comparison expressions.

- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)
- Not equal (!=)
- Equals (==)

They can be used in if and else statements like so (all are true statements):

```
if(1<2)
if(1==1)
if(1<=1)
if(2!=1)
```

## Print Statements

Any expression can be printed like so:

```
print(exp)
```

## For loops

For loops can be used to iterate through lists.

```
var lst = [1, 2, 3]

for( i in lst ) {
    print(i)
}
```

## Functions

Functions are declared with a body like so:

```
function foo(){
    print("hello")
}
```

Functions can be called with function call statements:

```
foo()
```

result:

```
hello
```

Functions can also be declared with any number of parameters

```
function foo(x : int, y : String) {  
  print(x)  
}
```

The function call for this function is:

```
foo(10, "hi")
```

Optionally functions can pass a value back to the origin of the function call using return statements.

```
function foo(x : int, y : String) {  
  return x  
}  
print(foo(10, "hi"))
```

## CatScript Grammar

---

```

catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            function_call_statement;

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
               '{', { statement }, '>';

if_statement = 'if', '(', expression, ')', '{',
              { statement },
              '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];

print_statement = 'print', '(', expression, ')'

variable_statement = 'var', IDENTIFIER,
                    [ ':', type_expression, ] '=', expression;

function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                      [ ':' + type_expression ], '{', { function_body_statement }, '>';

function_body_statement = statement |
                        return_statement;

parameter_list = [ parameter, { ',', parameter } ];

parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [, expression];

expression = equality_expression;

equality_expression = comparison_expression { ("!=" | "==") comparison_expression };

comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };

additive_expression = factor_expression { ("+" | "-" ) factor_expression };

factor_expression = unary_expression { ("/" | "*" ) unary_expression };

unary_expression = ( "not" | "-" ) unary_expression | primary_expression;

primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |
                    list_literal | function_call | "(" , expression, ")"

list_literal = '[', expression, { ',', expression } ']';

function_call = IDENTIFIER, '(', argument_list , ')'

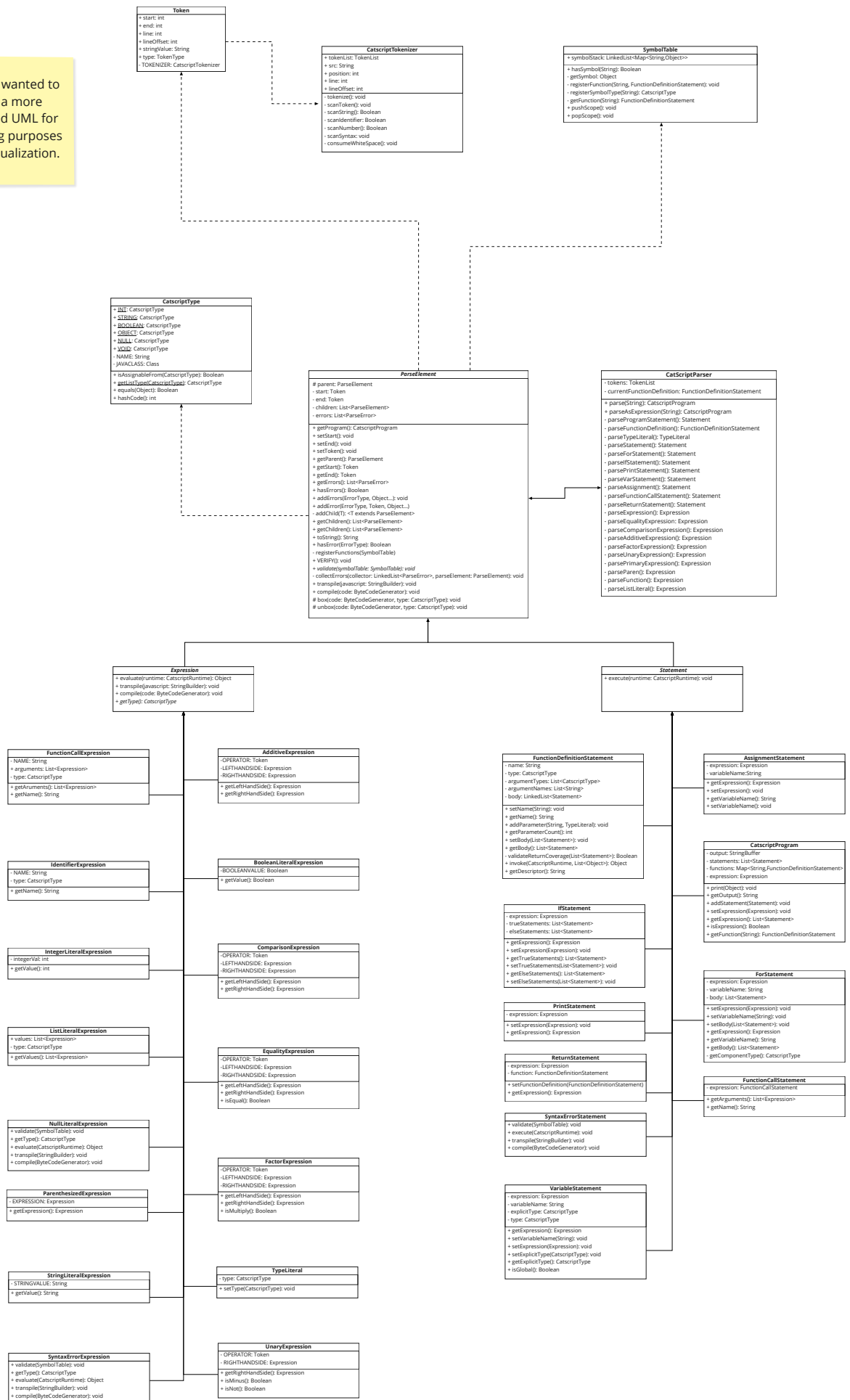
argument_list = [ expression , { ',', expression } ]

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' , type_expression, '>']

```

## Section 5: UML

Note: I wanted to use a more detailed UML for learning purposes and visualization.





## Section 6.

To begin our compiler we chose to use recursive descent as the primary design for Catscript. Recursive descent is very straightforward to implement but it does require time since it is written from scratch. Another route we could have taken to design our Catscript compiler would be to use a parser generator. A parser generator would automate the process of parsing and tokenizing where we would not have to implement any code for the tokenizer and parser. The reason for this might be to speed up development time in theory, but there are still some flaws to using a parser generator. The parser generator uses regular expressions to compile our language and this would be hard to read from a developers point of view. The choice to use a parser generator was not made due to the fact that we would overcomplicate the development process with complex debugging and regular expressions, hence the reason for using recursive descent.

## Section 7.

For the Catscript project we used test driven development. This was extremely useful for us during the development cycle of Catscript. Many tests were run to check the validity of our compiler. These tests were split across the checkpoints (tokenizer, parser, evaluations, and bytecode). After the completion of a checkpoint, we ran many tests designed to check if our implementation was correct. This made for clear direction and easy debugging along the way. Test driven development for this project was a very strong solution for creating a valid compiler and I would recommend this strategy for the future.