

Jacob Connelly

CSCI 468

May 1, 2022

Capstone Submission

Section 1: Program

Program source code can be found in src.zip located in the same directory of this document.

GitHub Link: <https://github.com/Jacob-Connelly/csci-468-spring2022-private/tree/master/capstone/portfolio>

Section 3: Teamwork

The teamwork portion of this project was to write three unit tests and CatScript documentation for each team member. My team consisted of a single teammate I will refer to as team member 1. I will be turning in team member one's unit tests located at `../test/java/edu.montana.csci.csci468/TeamMemeberTests.java` and the documentation he wrote located in section four of this document. Team member one spent approximately 5 hours writing the documentation and unit tests. I also spent about 5 hours writing the documentation and unit tests for him. Developing CatScript and getting all tests to pass took me about 20 hours.

Section 3: Design Pattern

```
package edu.montana.csci.csci468.parser;

import java.util.HashMap;
import java.util.List;
import java.util.Objects;

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;
}

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;
}

@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) {
        if (type == NULL) {
            return true;
        } else if (type instanceof ListType) {
            ListType otherList = (ListType) type;
            return this.componentType.isAssignableFrom(otherList.componentType);
        }
        return false;
    }

    public CatscriptType getComponentType() {
        return componentType;
    }

    @Override
    public String toString() {
        return super.toString() + "<" + componentType.toString() + ">";
    }
}
}

```

One design pattern we used during the development of CatScript is the Memoization pattern. This is a pattern that is used for program optimization where the results of an expensive call are stored in a cache for quick lookup when needed. When the method is called again with the same input, the results are returned from the cache instead of invoking the method again. In CatScript Memoization happens in the CatscriptType class on lines 36-44 in the getListType method. This method returns a CatscriptType object by invoking a static class called ListType. Before Memoization a new instance of ListType would be called and operations performed to get the correct CatscriptType for the list. Memoization is implemented by creating a HashMap where CatscriptType is the key and ListType is the value. This global HashMap is considered the cache. Every time getListType is called we will attempt to find the CatscriptType from the cache and return it if found, otherwise a new ListType will be made and inserted into the cache. Memoization efficiently optimizes the getListType method by only creating a new ListType if that CatscriptType has not been used before. It cuts down on the amount of object creations thus optimizing the application.

Section 4: Technical Writing

Catscript Documentation

Catscript Documentation	3
Introduction	4
Comments	5
Type System	6
Types	6
Variable Type Inference	6
Lists	7
Operations & Expressions	7
Equality Expression	7
Comparison Expression	8
Additive Expression	8

Factor Expression	9
Unary Expression	9
Primary Expression	10
Functions & Control Flow	10
For Statement	10
If Statement	10
Print Statement	11
Variable Statement	11
Assignment Statement	12
Functions	12

Introduction

Catscript is a simple scripting language. It was created for educational purposes and is intended to be used by students who are creating recursive descent compilers to compile Catscript to Java bytecode in the CSCI 468 compilers course. The complete catscript grammar is included for reference below.

```
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, '>' ]
```

Comments

Comments are an important part of development and allow for inserting explanations or notes into the code. Anything on the same line after the `//` symbol is ignored.

```
1 var num = 10
2 //num = 20
3 print(num)
```

Type System

Types

Catscript has six types:

- int: 32 bit integer
- string: a java style string
- bool: a boolean value, true or false
- list: a sequence of values
- null: a type representing the absence of a value
- object: any type of value

Variable Type Inference

Catscript supports two kinds of variable type assignment, explicit and implicit. Explicit typing is done by following the variable name with a colon and then one of the catscript type reserved keywords seen above. Implicit typing assigns the variable the type of the right hand side expression. Variables are statically typed and cannot be assigned to a different type.

```
1  var num = 10
2  var num2 : int = 10
3
4  num = "10"
5  num2 = "10"
6
```

Parse Errors Occurred:

```
Line 4:num = "10"  
      ^
```

Error: Incompatible types

```
Line 5:num2 = "10"  
      ^
```

Error: Incompatible types

Lists

A list is a statically typed variable which contains a collection of expressions. They can be explicitly declared or infer their type from the type of the items in the list. List items must be all of the same type. Object types can be used to store values of different types.

```
1 var myList:list<int> = [1, 2, 3] //explicit type declaration  
2 var myList2:list<string> = ["1", "2", "3"]  
3  
4 var myList3 = [1, 2, 3] //implicit type declaration  
5
```

Operations & Expressions

Expressions are symbols and operations that evaluate to a value.

Equality Expression

The equality expression evaluates to a boolean true or false value depending on the equality of the comparison of the left and right hand side expression and the operator used.

	!=	==
True	Expressions are not equal	Expressions are equal
False	Expressions are equal	Expressions are not equal

```

1 print(1 != 2) //true
2 print(2 != 2) //false
3
4 print(1 == 1) //true
5 print(1 == 2) //false
6
7

```

Comparison Expression

The comparison expression evaluates to a boolean true or false value depending on the comparison of the left and right hand side expression and the operator used.

	>	<	>=	<=
True	Left greater than right	Left smaller than right	Left greater or equal to right	Left smaller or equal to right
False	Left smaller or equal to right	Left larger than right	Left smaller than right	Left greater than right

```

1 print(1 > 2) //false
2 print(1 > 1) //false
3 print(2 > 1) //true
4
5 print(1 < 2) //true
6 print(1 < 1) //false
7 print(2 < 1) //false
8
9 print(1 >= 2) //false
10 print(1 >= 1) //true
11 print(2 >= 1) //true
12
13 print(1 <= 2) //true
14 print(1 <= 1) //true
15 print(2 <= 1) //false
16
17

```

Additive Expression

The additive expression is used to add integer values using the + operator and subtract them using the - operator. The + operator is also overloaded to combine strings. If the + is overloaded to concatenate a

string with a null, boolean, or integer value, the other type value will be converted to a string and combined.

```
1  var num = 5
2
3  num = num - 2
4  print(num) //3
5
6  num = num + 2
7  print(num) //5
8
9  var hello = "hello"
10 var world = "world"
11
12 print(hello + world) //helloworld
13
14
```

Factor Expression

The factor expression is used to multiply integer values using the * operator or divide them using the / operator. Note that floats are not supported so the returned value is rounded up or down.

```
1  var num = 8
2
3  num = num / 2
4  print(num) //4
5
6  num = num * 2
7  print(num) //8
8
9  print(38/5) //7
10 print(34/5) //6
11
12
```

Unary Expression

The unary expression is a one sided expression which uses the not operator to negate the expression.

```
1 var alwaysTrue = true
2
3 print(not alwaysTrue) //false|
4
```

Primary Expression

Primary expressions are the lowest level of the expression grammar and include identifiers, string literals, integer literals, the boolean literal true and false values, the null literal value, list literals, function calls, and parenthesized expressions.

```
1 "hello"
2 1
3 true
4 false
5 null
6 varName
7 [1, 2, 3]
8 (1 > 2)
9 myFunc(1, 2)|
```

Functions & Control Flow

Statements perform an action. They have side effects which interact with the execution environment.

For Statement

The for statement is a statement that is used for looping. It is useful for iterating through lists or performing a statement or expression located in the body of the loop a number of times.

```
1 for (x in [1, 2, 3]){
2     print(x) //output: 1 2 3|
3 }
```

If Statement

The if statement is a conditional gate. The expression in the if statement needs to be true for the statement or expression in the body to be executed.

```

1 var a = 1
2 var b = 2
3
4 if (a > b) { //false
5     print("Hello") //will not print
6 }
7
8 if (a < b) { //true
9     print("World") //will print "world"
10 }
11

```

The If statement also has an optional else clause that will execute the code within its body if the if expression is false. While else-if statements are not natively supported, you can achieve the same effect by chaining if statements.

```

1 var a = 1
2 var b = 2
3
4 if (a > b) { //false
5     print("Hello") //will not print
6 } else {
7     print("Hi ") //will print "Hi "
8 }
9
10 if (a < b) { //true
11     print("World") //will print "world"
12 }
13

```

Print Statement

The print statement is a simple statement that sends the expression inside its surrounding parentheses to the standard output.

```

1 for (i in [1, 2, 3, 4, 5]){
2     i = i + 1
3     print(i) //output: 2 3 4 5 6
4 }

```

Variable Statement

The variable statement associates a value with an identifier. The type of the variable can be inferred or explicitly defined. For more information see the type system section.

```

1 var num = 5
2 var num2:int = 5
3
4 var str = "hello"
5 var str2:string = "hello"
6
7 var boolean = true
8 var boolean2:bool = true
9
10 var obj = "This is an object"
11 var obj2:object = "This is an object"
12
13 var myList = [0, 1, 2, 3]
14 var myList2:list<int> = [0, 1, 2, 3]

```

Assignment Statement

The assignment statement is used for reassigning a variable to another value.

```

1 var one = 1
2 one = 2
3 print(one) //output: 2

```

Functions

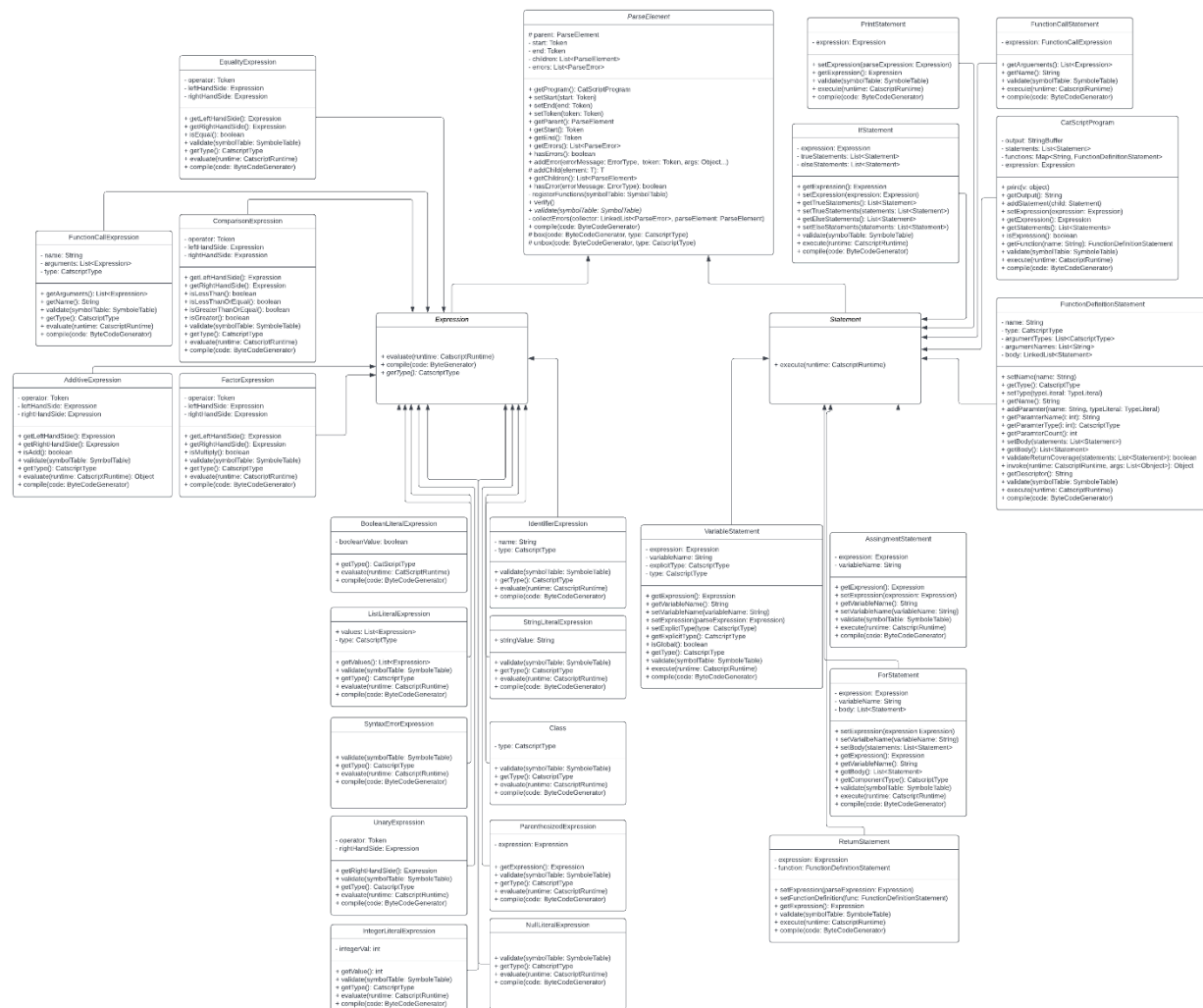
A function is a piece of code that you want to reuse. There are two parts to a function, declaring it and calling it. Declaring a function is where you define what the function does and what it's called and is dictated by the function definition statement. Calling a function is where the code is actually executed and it is dictated by the function call statement, which is the identifier of the function followed by values you want to pass in to the function. The placeholder for these values in the function definition are called parameters. Each parameter can specify a specific type. The return type of the function can also be specified by adding *: type* after the function identifier. Additionally, the return statement will return a value back to the point in executing the program where the function was called.

```

1
2 var one = 4
3 var two = 6
4 function myFunc(in1:int, in2:int){
5     return in1 + in2
6
7 }
8
9 print(myFunc(one, two)) // prints 10
10

```

I created the below UML diagram to represent the parse elements of this project. As depicted, ParseElement is an abstract class containing the CatScriptProgram and various methods required to parse and run the program. Expression and Statement are both abstract classes that inherit from ParseElement. Each different Expression and Statement defined in the CatScript grammar are concrete implementations that inherit from Expression and Statement respectively.



Section 6: Design Trade-offs

A design trade-off made while designing CatScript is the decision to create a recursive decent parser by hand instead of using a parser generator such as ANTLR. While Carson Gross made this decision for us, I believe he made the correct choice. Many modern-day languages use recursive decent for their parsers such as C#. Handwriting a recursive decent parser allowed us to fully understand how recursive decent worked by giving concrete examples. Many parser generators create a recursive decent parser but with code that is near impossible to debug. In class Carson Gross showed us a parser generated from ANTLR that contained multiple classes with 1000+ lines. Many methods in these classes were extremely difficult to understand and used variable names such as 'a' with no description of the methods operations. I believe Carson Gross made the correct choice in having us hand write a recursive decent parser because it was simpler to understand and much easier to debug.

Section 7: Software Development Life Cycle Model

Test Driven Development (TDD) was used when developing CatScript. TDD is a software development model where software requirements are converted to test cases before any business logic has been created. Throughout the development of the application, the tests are continuously run against the application to confirm that all requirements have been met. In the case of CatScript Carson Gross gave us a test suite that covers most CatScript requirements. We developed against these test cases until all tests passed. TDD is a very useful tool while developing because I was able to understand the exact required result that each method/process needed to return. It kept my code on the correct path and provided a valuable way to debug problematic code. On the other hand, TDD comes with its faults. I found that the test suite did not cover every scenario and I was able to get all tests to pass missing crucial portions such as not implementing the AssignmentStatement execute() method. I believe that TDD can provide a false sense of security when it comes to testing code. It can feel like every scenario and every edge case has been covered but some bugs will inevitably fall through the cracks. This is why a well laid out testing plan is required with TDD to find any issues that were not covered in the test suite. When a new bug is found a test should be written for the new scenario in order to

continuously improve your test suite. Overall, I believe that TDD works very well in an educational setting and helped me understand testing more in depth.