

# Capstone Portfolio

William Walkuski

Tester: Jack Tetrault

CSCI 468: Compilers

Spring 2022

Professor: Carson Gross

# Section 1: Program

The source code is found in source.zip in the same directory. It is written in Java using the IntelliJ IDE. The codebase was continually managed using GitHub.

Source code found at:

/Users/WillW/csci-468-spring2022-private/capstone/portfolio/source.zip

Or

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

## Section 2: Teamwork

### Member Contributions

Team member 1 was responsible for programming and implementing full functionality of the project. Features implemented include tokenization, parsing, eval, and compilation of the Catscript language. Functions were implemented to satisfy the provided test suites.

Team member 2 provided additional tests to run the program against to ensure functionality in cases not originally covered in the base tests and to continue to add to the test-driven development cycle. They also provided the full technical documentation of the Catscript language which is included in the next section.

### Time Estimates

Team member 1: 80 hours (~89%)

Team member 2: 10 hours (~11%)

## Section 3: Design Pattern

A design pattern included in this project is memoization. Memoization is a technique used to optimize a program by reducing expensive function calls. To eliminate these function calls, a cache is created with a map. If the function is called and the desired result is already stored in the cache, that value is returned. Otherwise, the cache is updated. Our implementation was not designed to be safe with multiple threads.

This implementation can be found in the CatscriptType.java class beginning on line 38. This file is found in src/main/java/edu.montana.csci.csci468/parser. The code block is also included below.

```
// Memoization implementation
static Map<CatscriptType, CatscriptType> CACHE = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType potMatch = CACHE.get(type);
    if (potMatch != null) { // check if the type already exists
        return potMatch;
    } else { // if not, add the new type
        ListType listType = new ListType(type);
        CACHE.put(type, listType);
        return listType;
    }
}
```

# Section4: Technical Writing

## Catscript Technical Documentation

### Introduction

This documentation will follow the grammar of CatScript providing descriptions and examples of each element that CatScript consists of.

### Features

#### CatScript Types

CatScript is a statically typed language with a simple type system. This means that once a type is defined it cannot be changed. The types include:

- int - a 32 bit integer
- string - java-style strings
- bool - boolean value
- list - a list of value with type 'x'
- null - the null type
- object - any type of value

#### Error Handling

Error handling within CatScript is similar to what you would expect from other languages. Line numbers and line offsets are stored to give specific feedback on where an error occurs.

#### Comments

Commenting in CatScript is identical to commenting in Java. Comments begin with `//` and anything after will be ignored by the compiler.

ex:

```
// some comment
```

#### Expressions

Expressions evaluate to some value, either object or literal types.

#### Type Expressions

As defined in the grammar:

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

### *Integer Literal*

Integer literals, as listed above, are 32 bit integers. CatScript does not support a decimal type like floats.

ex:

```
50
```

### *String Literal*

String literals are arrays of characters of any length. String literals must be enclosed in double quotes. String concatenation falls under the additive expression.

ex:

```
"Hello world"
```

### *Boolean Literal*

Boolean literals behave very similarly to other languages. they evaluate to either true or false.

ex:

```
true  
false
```

### *Null Literal*

Null literal expressions are objects with no value.

ex:

```
null
```

### *List Literal Expressions*

As defined in the grammar:

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

List literals are very similar to arrays, they can contain values of any type supported by CatScript. They must be enclosed in brackets and values must be delimited by commas.

ex:

```
[1, 2, 3, 4, 5]
```

Lists may contain values of varying types.

ex:

```
[1, "Hello", true, 2, "World", false]
```

## Parenthesized Expressions

Parenthesized expressions are any expression surrounded by parentheses. Because of the recursive nature of the grammar, expressions can be surrounded by any number of parentheses pairs and will evaluate to the same value.

ex:

```
(expression)
(50)
("Hello world")
((((50))))
```

## Equality Expressions

As defined in the grammar:

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

Equality expressions check if two expressions are equivalent or not and return a boolean value based on the result. '==' is used for checking if two expressions are equal and '!=' is used for checking if two expressions are not equal.

ex:

```
1 == 1
1 == 2
1 != 1
1 != 2
```

These evaluate to:

```
true
false
false
true
```

## Comparison Expressions

As defined in the grammar:

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

Comparison expressions check a condition and return a boolean value based on the result. Comparisons can be '<', '>', '<=', or '>='.

ex:

```
1 < 2
1 > 2
```

```
1 <= 1
1 >= 1
```

These evaluate to:

```
true
false
true
true
```

## Additive Expressions

As defined in the grammar:

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

Additive expressions include adding or subtracting integer values. String concatenation can also be done with this expression using the '+' operator with a string on the left, right or both sides. Parentheses can be used to enforce precedence.

ex:

```
1 + 1
1 + 1 + 1
2 - 1
(1 + 3) - 2
"Hello " + "world"
"Hello world" + 1
```

These evaluate to:

```
2
3
1
4
Hello world
Hello world 1
```

## Factor Expressions

As defined in the grammar:

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

Factor expressions behave very similarly to additive expressions but encompass multiplication and division. Parentheses can again be used to enforce precedence.

ex:

```
1 * 2
1 * 2 * 3
6/2
2 * 8/(2 + 2)
```



These evaluate to:

```
2
6
3
4
```

## Unary Expressions

As defined in the grammar:

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

Unary expressions precede values and typically convert to opposites using '-' or 'not'. '-' is used before integers and 'not' is used before booleans. These can be stacked many times.

ex:

```
-50
--50
not true
not not true
```

These evaluate to:

```
-50
50
false
true
```

## Primary Expressions

As defined in the grammar:

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

## Function Call

As defined in the grammar:

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

Function calls consist of an identifier (name) followed by an argument list surrounded by parentheses.

ex:

```
foo(1, 2, 3)
```

## Argument Lists

As defined in the grammar:

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

Argument lists are any number of expressions delimited by commas.

ex:

```
1, 2, 3
```

## Statements

### For Statement

As defined in the grammar:

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

For statements traverse lists with iteration. They follow the following format.

ex:

```
for(varName in list) {  
    // body  
}
```

In practice:

```
for(x in [1, 2, 3, 4, 5]) {  
    print(x)  
}
```

Outputs:

```
1 2 3 4 5
```

### If Statement

As defined in the grammar:

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

If statements execute specific code blocks based on a conditional value. they follow the following format.

ex:

```
if(expression) {  
    // execute this code if condition is met  
}
```

In practice:

```
var x = 10
if(x == 10) {
    print(x) // x would be printed since it meets the condition
}
```

If statements can also have else clauses that follow.

ex:

```
if(expression) {
    // execute this code if condition is met
} else {
    // execute this code if condition is not met
}
```

In practice:

```
var x = 10
if(x == 10) {
    print("x is 10")
} else {
    print("x is not 10")
}
```

## Print Statement

As defined in the grammar:

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

Print statement use the print command directly followed by parentheses. Whatever is inside the parentheses will be output.

ex:

```
print("Hello world")
```

This evaluates to:

```
Hello world
```

## Variable Statement

As defined in the grammar:

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

Variable statements assign a variable name to a value. They follow the following format.

ex:

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

Notice the type (following the colon) can be omitted and an object type will be assigned by default.

ex:

```
var x = 10
```

### Function Call Statement

As defined in the grammar:

```
function_call_statement = function_call;
```

Function call statements are the same as function call expressions.

### Assignment Statement

As defined in the grammar:

```
assignment_statement = IDENTIFIER, '=', expression;
```

Assignment statements assign a new value to an existing variable

ex:

```
var x = 10  
x = 22  
print(x)
```

This evaluates to:

```
22
```

Note that CatScript is statically typed so variable types cannot change throughout execution. The following example is invalid and would result in a parse error.

ex:

```
var x = 10  
x = "Hello"  
print(x)
```

### Function Declaration

As defined in the grammar:

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +  
                        [ ':' + type_expression ], '{', { function_body_state  
ment }, '}';
```

Functions are good for easily calling specific code blocks that will be repeatedly used. To declare a function the format is as follows.

ex:

```
function funcName(parameters : parameterType) : returnType {  
    // body of function  
}
```

In practice:

```
function helloWorld(x : string){  
    print(x)  
}  
  
helloWorld("Hello world")
```

Note that a function can have zero or more parameters. Parameters and return may or may not have a defined type. If the return type is omitted, 'void' is implied by default.

### Function Body Statement

As defined in the grammar:

```
function_body_statement = statement |  
                        return_statement;
```

The body contained in a function consists of statements or expressions.

### Parameter List

As defined in the grammar:

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

A parameter list is a list of parameters delimited by commas.

### Parameter

As defined in the grammar:

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

A parameter is an identifier optionally followed by a colon and a type.

ex:

```
x : string
```

### Return Statement

As defined in the grammar:

```
return_statement = 'return' [, expression];
```

Return statements return specified values from functions.

ex:

```
function funcName(parameters : parameterType) : returnType {  
    // body of function  
    return returnval  
}
```

In practice:

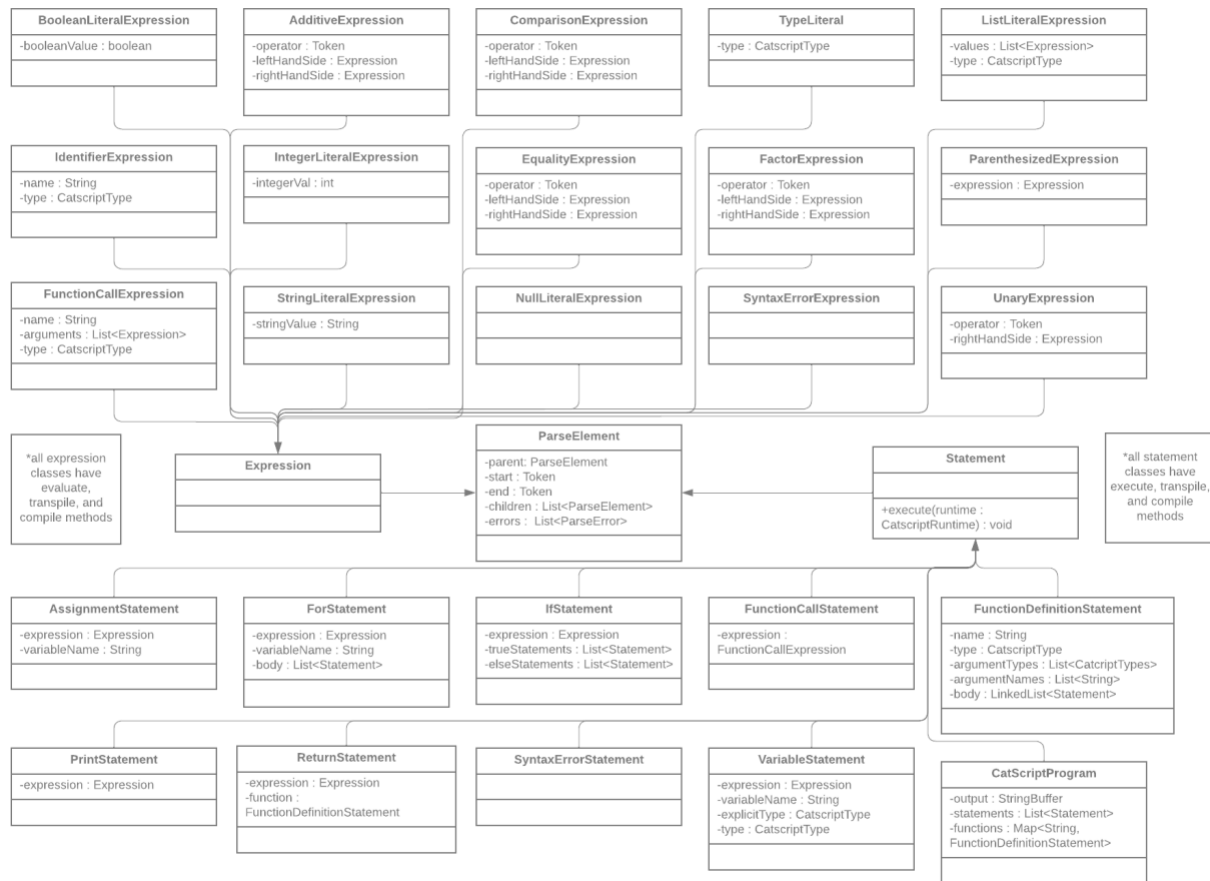
```
function helloWorld(x : string) : string{  
    return x  
}
```

```
a = helloWorld("Hello world")  
print(a)
```

This would output:

```
Hello world
```

## Section 5: UML



There was no UML creation prior the start of this project based on the way the class was formatted. Above is an expansive class diagram stemming from the *ParseElement* class. It illustrates how elements to be parsed fall under either an expression or a statement. From there it shows all the different expressions/statements that could be executed upon along with their respective attributes. While this is only a fraction of the codebase, it covers the bulk of the most important part of the compiler. All together they provide the framework for the parsing stage. These classes also include evaluate or execute, transpile, and compile methods as mentioned in the notes. Obviously more detail could have been included for each class but at this scale it was omitted for readability reasons and to allow a broader view of this portion of the codebase.

# Section 6: Design Tradeoffs

## Recursive Descent

Prior to beginning work on this project, the decision was made to use a recursive descent parsing algorithm, the method used for most production compilers, as opposed to a parser generator. As with any choice like this, there are pros and cons to each option. This section will cover why recursive descent was chosen as the preferred design choice.

Recursive descent is a Top-Down Parser meaning that it builds a parse tree from the top and down. The grammar is traversed recursively starting from the top until a terminal is hit. If non-terminals still exist, the top of the grammar is called again until all branches are terminals. To do this, a grammar that eliminates left recursion and left factoring is required. This technique fit well with the grammar that we were working with.

Recursive descent is known to be a simpler method of writing a compiler despite there generally being more code required. Although there is more code, since it is simpler than a parser generator the code ultimately is more readable, more maintainable, and therefore also easier to debug. It also provides a better understanding of the recursive nature of grammars. Parser generators are more standard among classes like this at other universities, but recursive descent is more applicable to production compilers.

## Avoiding Visitor Pattern

A common technique that we decided to avoid is known as the Visitor Pattern. This idea consists of placing new behavior or operations into a separate class rather than introducing modifications to an existing object structure. Instead, we decided to put eval, transpile, and compile directly onto the parse tree nodes. Generally, these very different operations would be separated into their own classes. This choice was made primarily to preserve as much simplicity as possible throughout the program. It keeps everything in one place and makes implementation more fluid.



## Section 7: Software Development Lifecycle

The software development lifecycle used in this project is test-driven development (TDD). We were provided with several test suites to verify that our code was producing the intended result to ultimately build a functioning compiler. There were dedicated test suites for each checkpoint. The majority of tests were provided prior to beginning work on the project however several more were included at the end by team member 2 to further verify that the code behaved as intended.

### How TDD Works

In theory TDD operates in 6 steps. These steps are as follows:

1. Add a test
  0. Requires focusing on requirements before writing code
2. Run all tests (new test should fail)
  0. Proves that new code is required for desired feature
3. Write the simplest code that passes the new test
4. All tests should now pass
  0. If tests fail, revise the new code until they pass. This ensures new code meets the requirements and does not break existing functionality.
5. Refactor as needed, using tests after each refactor to ensure functionality is preserved
  0. This improves readability and maintainability without breaking functionality
6. Repeat

### Our Experience with TDD

As mentioned above, the majority of tests were provided from the beginning as opposed to adding them as we go. They were split into 4 sections. One for tokenization, one for parsing, one for eval, and one for bytecode generation. These tests not only verified that correct syntax passed but that incorrect syntax was also caught appropriately. This ultimately was a clean and efficient way of doing things. It ensured that all test cases were covered, while still maintaining functionality as we went through the tests. We also found that having tests made debugging the functions much quicker and intuitive. The tests provided valuable error handling within a controlled case. The tests provided a clear direction for the project and promoted independent work. Having a clear answer of when things were working correctly or not and when all requirements were met made work rewarding and inspired confidence in the code being produced.