

Portfolio  
CSCI-468: Compilers  
Spring 2022

Moiyad Alfawwar  
Philip Ghede (Partner)

2022-05-06

## Section 1: Program

The source of the project is linked in the path of the following directory.  
`/capstone/portfolio/source.zip`

## Section 2: Teamwork

Our team is composed of two members. Team member 1: Moiyad Alfawwar, Team member 2: Philip Ghede. Team member 1 his primary contributions were the implementations of the tokenizer, parser, evaluation, and bytecode. Team member 1 spent approximately 90% of the time working on the project. Team member 2 contributed primarily by providing tests and documentation to the project to check if the functionality is implemented correctly and document the features of the Catscript programming language. Team member 2 spent approximately 10% of the time.

## Section 3: Design Pattern

In this project, I decided to use the memoization design pattern. The memoization design pattern is used to optimize the runtime of the program while running specific methods. In this instance, I memoized the `getListType` method. If the function is called and the list type exists in the cache created as a `HashMap`, then we return the one that we have saved in the cache instead of creating a new `ListType(type)` every single time the method is called. This should cut down some of the time complexity of the runtime of the program.

```
static final HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    // getting list-type from the cache called it memory.
    ListType listTypeMem = cache.get(type);
    // if the memory is null, means that it hasn't been cached yet.
    if (listTypeMem == null) {
        // create the type and put it into the cache and return it.
        ListType listType = new ListType(type);
        cache.put(type, listType);
        return listType;
    } else {
        // if It's cached return the cached type.
        return listTypeMem;
    }
}
```

## Section 4: Technical Writing

### Catscript

In this course we will be creating a small programming language called CatScript

### 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 , '>']

```

## CatScript Types

CatScript is statically typed, with a small type system as follows

- 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

## Features

### Arithmetic Operators

- Addition operator +
- Subtraction operator -
- Multiplication operator \*
- Division operator /

```
print(1+1)
print(1-1)
print(2*3)
print(1/1)
```

Output:

```
2
0
6
1
```

### Variables

```
var x : string = "Howdy!"
var x = 25
```

### Lists

Declaring and assigning lists.

```
var myList = [1,2,3]
var myList = [1,"string", 3] //lists behave covariantly
```

Integer lists

```
var myList : list<int> = [1,2,3]
```

String lists

```
var myList : list<string> = ["a", "b", "cde"]
```

Object list

```
var objLst : list<object> = ["apple", true, 1, "object"]
```

### For Loops

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

Output:

```
1  
2  
3
```

## Comparison

Catscripts supports Equality expressions, Comparison expressions

Equality operators:

Equal == and Not Equal !=

```
var x : bool = (7==7)  
var y : bool = (7!=7)  
print(x)  
print(y)
```

Output:

```
true  
false
```

Comparison operators:

Greater than >, Greater than or Equal >=, Less than <, and Less than or Equal <=

```
var x : bool = (10>9)  
print(x)  
var y : bool = (10>10)  
print(y)
```

Output:

```
true  
false
```

## If Statements

```
var x : int = 1  
var y : int = 2  
if(x == y){  
  print("same")  
} else if(x != y) {  
  print("not the same")  
}
```

```
} else {  
    print("fail")  
}
```

Output:

not the same

## Printing

```
print("Howdy!")
```

Output:

Howdy!

```
var strX : string = ("Howdy, ")  
var intY : int = 10  
  
print(strX + intY)
```

Output:

Howdy, 10

Printing boolean expressions:

```
print(1==1)  
print(80>90)
```

Output:

```
true  
false
```

## Functions

Function with parameter:

```
var x = "Howdy!"  
function foo(str) {  
    print(str)  
}  
  
var x = "Howdy!"  
function foo(str : string) {  
    print(str)  
}
```



Function call:

```
foo(x)
```

Outputs:

Howdy!

Function without any parameters:

```
function foo(){  
  print("A function without parameters")  
}  
foo()
```

Output:

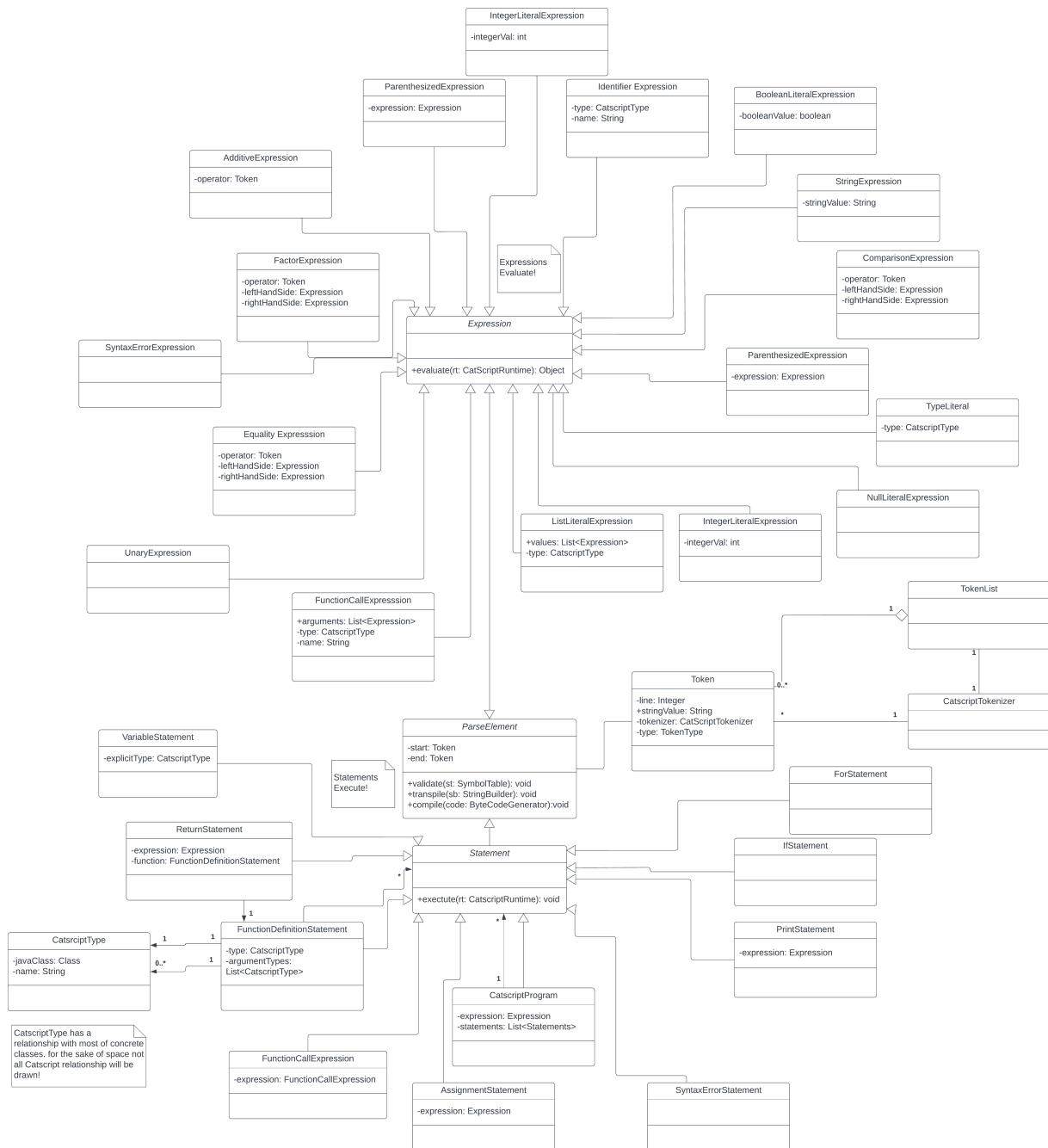
A function without parameters

## Comments

Catscript support comments.

```
// Catscript Comment
```

## Section 5: UML



## Section 6: Design Trade-Offs

This project was created using recursive descent parsing, it is also called hand-crafted parsers. Recursive descent has many advantages over parse generators. Features of recursive descent include speed, it is faster than parse generators. Unlike parse generators, recursive descent is easy to understand. It is also much easier to modify, add features to, maintain and debug. One of the things that we did in the project is adding meaningful errors and helping the user debug their code faster. Of course recursive descent is not better in everything than parse generators. Parse generators require less code to get working and less infrastructure. Indeed that comes with the tax of being much harder to read, far from the hardware.

Another trade off, that this project constructed in an unusual composition. The compiler, the transpiler, and evaluation are all embedded within the parser. This makes it simpler and easier to code all in one place. However, the lack of separation of concerns can lead to unorganized and tightly coupled which would eventually make the code base less maintainable in the long run even though we are using recursive descent.

In my opinion, what makes recursive descent a much better approach is that reflects the recursive nature of the programming language grammar directly, despite having the footprint of the code base larger than it would be if it was done using parser generator.

## Section 7: Software Development life cycle model

In this project we used a Test Driven Development life cycle model. In this model, we plan tests of the expected behavior of the required functionality of the project. This allows me to complete the development process and achieve an accurate output to the requirements. The tests themselves do not only test the required output functionality, but also try to tackle errors and when the program should stop. For instance, if a user is creating a list and the user's syntax is not inline with the grammar of Catscript it should halt and give out an error message. Therefore, this model has a great workflow. In fact this project may make me learn the test kits of a new language that I am learning to create projects in.

Since the tests are usually created in sections and each one ensures specific parts of the program are behaving and working correctly. This allowed me to focus and be on track of each functionality. Often, not always, the errors allow me to know where exactly they happened during the runtime process. Therefore, I know where exactly to start debugging. I think this test driven development life cycle goes hand in hand with debugging in my experience.

Of course it isn't all nice there are obviously some drawbacks, it takes a long time to create tests that are very valid to the task that needs to be done. Some portions of the tests in this project do not always provides good errors in this case some of the Bytecode tests were much different than the rest of the testing suit. The messages that I get from the Bytecode tests are not always helpful. That is probably not due to the life cycle model. It was just of the nature of Bytecode and its error messages.