

Montana State University

Catscript Compiler

Mark Mousel

## Teamwork

For this project, I was the primary programmer. Using the infrastructure code provided to me by Professor Carson Gross, I implemented a fully functional tokenizer, parser, and compiler for the toy language, Catscript. To ensure the code I wrote was working as intended and provided full functionality, Professor Gross also provided an extensive test suite that I used throughout my development process.

My partner on this project, Elliott Pryor, had two responsibilities. The first of which was to write the technical writing portion of this capstone assignment, the documentation for Catscript. His secondary responsibility was to expand the testing suite provided by adding three of his own tests.

## Time Estimates

Total estimated hours: 44 hours

Member 1:

Contributions: Primary programmer, Portfolio writer

Total estimated hours: 40 hours - 91% of the total estimated hours

Member 2:

Contributions: Technical writer, Unit tester

Total estimated hours: 4 hours - 9% of the total estimated hours

## Design Pattern

```
private static Map<CatscriptType, ListType> cache = new HashMap<>();

private static Map<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType l = cache.get(type);
    if(l != null){
        return l;
    }
    l = new ListType(type);
    cache.put(type,l);
    return l;
}
```

Above I have included a code snippet from the CatscriptType class that implements the memoization pattern. In short, this pattern works by placing the list type into the hash map titled cache. When the list type is asked for later, it can simply be pulled out of the cache instead of having to run the getListType() function every time. This is a simple, but very useful optimization that helps to reduce the runtime of the compiler.

# Catscript Documentation

## 1 Introduction

Catscript is a simple scripting language. Here is a basic hello world example:

```
1   var x = "hello world"
2   print(x)
```

Catscript overall is fairly simple, but it does offer basic functionality of a scripting language. For instance, there are for and while loops, branching control (if), and functions. Catscript is built on Java, and has many similarities with the Java language.

It is possible to do comments in Catscript, where “//” starts a single line comment and “/\* \*/” is a multi line comment.

```
1   var x = "hello world" // this is a single line comment
2   /* if we want a multi line comment
3   we can use this.
4   This is useful for code documentation in Catscript */
5   print(x)
```

## 2 Datatypes in Catscript

Catscript is statically typed, but it supports type inference with the ‘var’ keyword.

- int - a 32 bit integer
- string - a java-style string
- bool - a boolean value
- list - a list of values of any Catscript type. <sup>1</sup>
- null - null value

---

<sup>1</sup>List objects in Catscript are immutable. Meaning after creation their values cannot be changed.

- object - any catscript type

The example below are both valid Catscript examples, both x and y evaluate to type string. Where y is given the explicit type string, and x infers the type based on the right hand side of the equation.

```
1   var x = "hello world"
2   var y : string = "hello world"
```

In this example, variable x (line 1) compiles correctly as it would infer type int, but variable y would error because you cannot assign an int to an explicitly typed string.

```
1   var x = 1
2   var y : string = 1
```

## 3 Language and Examples

### 3.1 Statements

#### 3.1.1 For Statement

In catscript it is possible to make a for loop to iterate over objects. Basic syntax is as follows.

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

In general it follows the form of 'for (VARIABLE in EXPRESSION)' where VARIABLE is any variable name (must not be declared elsewhere) and EXPRESSION is a list literal or an expression that returns a list (i.e a function call that returns a list).

#### 3.1.2 If Statement

If statements are used for logical branching. They are similar to java if statements. There is no logical and nor logical or in Catscript, so this functionality has to be handled through nested if, else if, else statements. In Catscript it is possible to do if (BOOLEAN EXPRESSION), else if (BOOLEAN EXPRESSION), else.

```
1   if (x == 1){
2
3   } else if (x == 2){
4
5   } else {
6
7   }
```

Where the else if and else statements are optional, but they cannot appear alone. I.e it is not possible to have an else or else if without first an if statement.

### 3.1.3 Print Statement

The print statement is a simple method that prints output onto standard out. It is equivalent to `System.out.println` in java, meaning that it always contains a newline character at the end of the print.

```
1   print("Hello world") // causes "Hello World\n" to print to console
2   print(1) // prints "1" to the console
3   print([1, 2, 3, 4]) // prints "[1, 2, 3, 4]" to the console
4   print(null) // prints "null" to the console
5   print(true) // prints "true" to the console
```

### 3.1.4 Variable Assignment

#### Variable Declaration

Variables are declared with the ‘var’ keyword Variables can either be implicitly typed or explicitly typed during declaration.

```
1   var x = 1 // implicitly typed as int
2   var y : int = 1 // explicitly typed as int
```

As noted in Section 2, giving an explicitly typed variable the wrong datatype causes an error because Catscript is statically typed.

#### Variable Assignment

Similar to most languages it is possible to assign or update a variable’s value using ‘=’

```
1   var x = 1 // implicitly typed as int
2   x = 2
```

### 3.1.5 Function Declaration and Calls

#### Function Declaration

The following lists a possible function declaration in Catscript.

```
1   function add(x, y : int) : int {
2       return x + y
3   }
```

In general, a function declaration follows: 'function' FUNCTION NAME + (any number of typed or untyped parameters, separated by ',') + a return type (if no type is given, it is assumed void) + '{' + function body + '}'

More function examples:

```
1    function add(x, y) : int {
2        return x + y
3    }
4
5    function concat(x: int, y: string) : string {
6        return x + y
7    }
8
9    function printList(x: list) {
10        for (y in x){
11            print(y)
12        }
13    }
```

## Function Calling

It is simple to call functions in Catscript. Simply just use the function name, and list the parameters surrounded by parenthesis.

```
1    function add(x : int, y : int) : int {
2        return x + y
3    }
4    add(1, 1) // output 2
```

## 3.2 Operations

### 3.2.1 Addition

**Mathematical:** Integer addition works the same as elementary math, it adds the left and right hand side together if they are both integers.

**Concatenation:** The '+' operator can also be used for concatenation. This is when two strings are joined together for one string. If one side of the '+' operator is any Catscript type while the other side is a string, then the other value is cast to a string and string concatenation is performed.

```
1    1 + 1 // outputs integer 2
2    "hello " + "world" // "hello world"
3    "trial number: " + 1 // "trial number: 1"
```

### 3.2.2 Subtraction

Subtraction can be performed between two integer values. It performs the elementary subtraction operation between the two values.

```
1      5 - 3 // 2
```

Also, for numeric operations. The '-' operator is overloaded to negate the number on the right hand side.

```
1      var x = -3 // -3
2      -x // 3
```

### 3.2.3 Multiplication and Division

These operations multiplication '\*' and division '/' perform the elementary multiplication and subtraction operations between two values. Special note for division, since integers are the only supported numeric type in Catscript, integer division is performed. This means that any decimal value is rounded *down* to the nearest integer.

```
1      2 * 2 // 4
2      8 / 2 // 4
3      9 / 2 // 4
```

### 3.2.4 Boolean

A number of boolean logic operators are supported. The most simple is the 'not' operator which flips any boolean value to its opposite. Also supported are a variety of equality and inequality operators.

- Equality: ==: '=='
- Not Equal to: ≠: '!='
- Greater Than: >: '>'
- Greater Than or Equal to: ≥: '>='
- Less Than: <: '<'
- Less Than or Equal to: ≤: '<='

```
1      not true // false
2      not false // true
3      1 == 1 // true
4      1 == 2 // false
```



```
5      1 != 1 // false
6      1 != 2 // true
7      1 > 1 // false
8      2 > 1 // true
9      1 >= 1 // true
10     2 >= 1 // true
11     1 >= 2 // false
12     1 < 1 // false
13     1 < 2 // true
14     1 <= 1 // true
15     2 <= 1 // false
16     1 <= 2 // true
```

### 3.2.5 Order of Operations

Catscript follows order of operations in mathematical expressions. I.e it evaluates multiplication and division first, then addition.

```
1      2 * 2 + 3 * 4 // 18
```

Optionally, Catscript supports parenthesis to enforce order of operations, being the value within the parenthesis is evaluated first and then other operations continue.

```
1      not (1 > 2) // true
2      3 * (2 + 2) // 12
3      3 * 2 + 2 // 8
```

## 4 Catscript Grammar

The Catscript grammar is formally defined (in CBNF) as:

```
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 , ')'

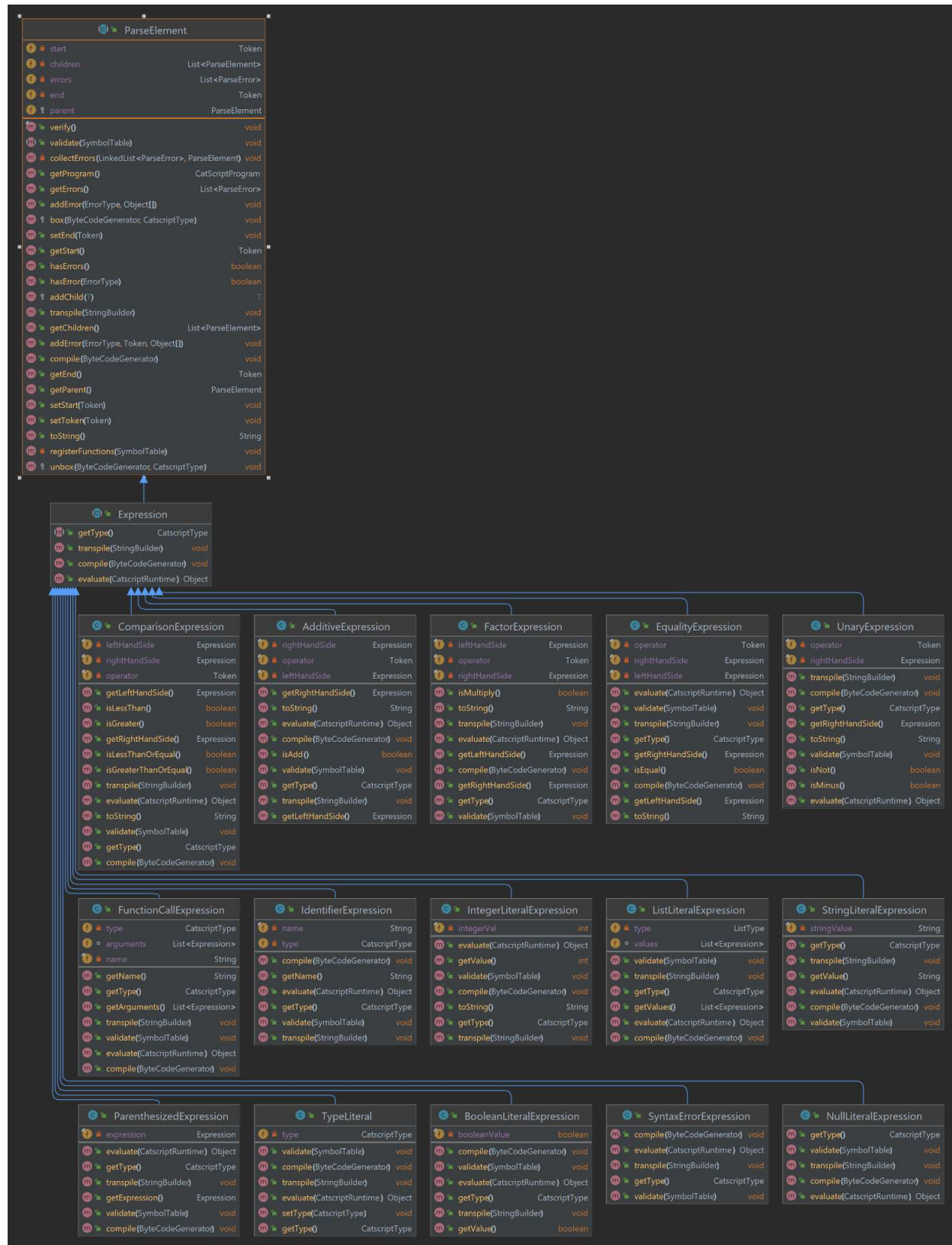
argument_list = [ expression , { ',', expression } ]

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

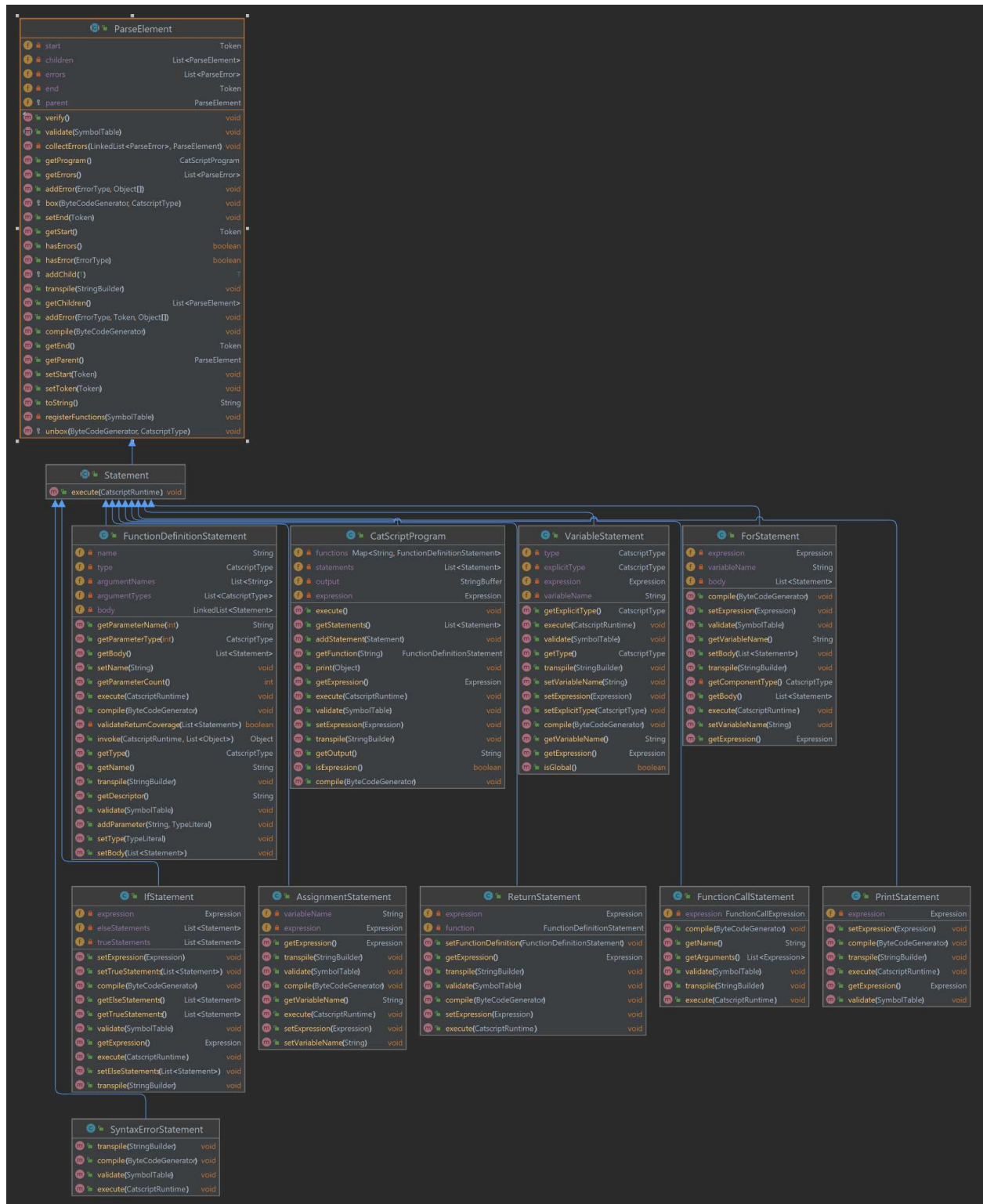
```

## UML

Within this Catscript compiler, parse elements are considered to be either expressions or statements. Expressions are defined as code that will evaluate to a specific value and statements are defined as blocks of code that execute. Below are two UML diagrams that illustrate this relationship of inheritance. The first diagram shows which classes are children of the Expression class, and the second shows which classes are children of the Statement class. Both diagrams show all elements as children of ParseElement.



Expression Inheritance UML



Statement Inheritance UML

## **Design Trade-offs**

For writing the parser, it was decided to use recursive descent instead of a parser generator. This decision was made for a couple of reasons. The first of which is that recursive descent lines up very well with the recursive nature of the Catscript grammar and grammar in general. This meant that it was relatively simple to reference the grammar and determine how each expression and statement needed to be implemented. The second reason recursive descent was chosen was that writing the code in this way was far simpler and easier to understand than the same code that would be written for a parser generator. That said, recursive descent is not without its flaws, and utilizing this strategy meant that a lot more code needed to be written by hand than if a parser generator had been used.

## **Software Development Life Cycle Model**

As mentioned previously, Professor Gross provided an extensive test suite to facilitate development on this project, meaning that the life cycle model used was Test Driven Development (TDD). In an academic setting such as this, TDD is a very effective way of informing the programmer of exactly what the requirements and expectations for the project are. Debugging was also made easier by the use of this method, as when an error occurred it was much easier to track down where in the codebase it had popped up. While this means that the code needs to be written to match the tests exactly and even a solution to a problem that works on its own may not pass a test, it drastically reduces the chance of a bug passing through the cracks and popping up later down the line.