

Montana State University

Capstone Paper

Kaylee Fong

Tester Partner: Caden Fong

Compilers - CSCI 468

Spring Semester 2022

Table of Contents

Section 1: Program	3
Section 2: Teamwork	3
Section 3: Design Pattern	4
Section 4: Technical Writing	5
Section 5: UML	10
Section 6: Design Trade-offs	11
Section 7: Software development life cycle model	11

Section 1: Program

<https://drive.google.com/file/d/1WO-S1TIdlOfFlhS1fLLK8on0GQt-IimI/view?usp=sharing>

Section 2: Teamwork

General Teamwork Information

For this capstone project there were two team members, team member 1, the author of this report, and team member 2, the tester for this project. For this project over 150 hours were spent to get it to completion. Something to note for this project is that it was mostly an individual project and team member 2 also wrote their own compiler where team member 1 for this project was the tester for them.

Team Member 1 (The Programmer)

Team member 1's Primary contributions were writing the compiler through test-driven development and this report. Team member 1 spent around 80 percent of the total project time, or around 120 hours, on this project.

Team Member 2 (The Tester)

Team member 2's Primary contributions were writing three tests (tests are located in PartnerTests.java in the test directory) for the compiler and also writing the documentation for Catscript which can be found in Section 4: Technical Writing of this report. Team member 2 spent around 20 percent of the total project time on this project, in other words around 30 hours.

Section 3: Design Pattern

The design pattern used in this project is called memoization or the flyweight design pattern. The pattern is used in the file “CatscriptType.java” in the the “src/main/java/edu/montana/csci/csci468/parser/” file path and can be seen below:

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

Shown highlighted above is the implementation of the memoization design pattern. The reason for doing it this way is because the Catscript list type can be used many times throughout a single program and is a costly object to create every time. Using this pattern more efficiently uses space as each instance of a ListType for a specific primitive type like int only gets created once. It also means that this operation is faster for each subsequent call as getting the ListType object is then only a simple lookup instead of an object instantiation.

Catscript Documentation of Features

Introduction

Catscript is a simple scripting language with the ability to evaluate or compile code written within it's grammar rules that appear as follows.

```
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, '>']

```

Features

Catscript features syntax for basic data structures and control flow.

Comments

Catscript comments are single inline statements ignored during compiling and interpretation and can be written as follows

```
// This comment will be ignored by the parser
```

Variables and Type Inference

Catscript is statically typed and supports the following data types natively: null, void, int, string, bool, and list. Values can be assigned to variables globally and locally, but can only be defined once for a given identifier. However, the value can be reassigned just without the var keyword.

Variables can be defined with or without a type explicitly but will be inferred if not given, and is semantically written as follows

```
var identifier : int = 1
identifier = 2
or
var identifier = 1
identifier = 2
```

Lists are immutable in Catscript and can be defined as follows

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

Basic Operators

Catscript includes basic operators for modifying and calculating using ints' as well as a unary operator for bool values. The binary operators use left associativity excluding the two unary operators which are right associative.

Addition in Catscript operates following the rules of basic addition and is overloaded to support string concatenation, and is written in the following ways

```
1 + 5
or
1 + "Hello World"
or
"Hello " + "World"
```

Subtraction can be written as follows

```
5 - 1
```

Multiplication can be written as follows

```
2 * 2
```

Division can be written as follows

```
16 / 4
```

Negation can be written in two ways for negating an int and a bool, they are written as follows

```
-4  
and  
not true
```

Comparison Operators

Catscript supports all the basic comparison operators for ints' and an equality operator for comparing referential equality, each resolving to a bool value.

Int comparison operators include less than, greater than, less than or equal to, and greater than or equal to operators and are written as follows

```
1 < 2  
2 > 1  
6 <= 8  
5 >= 4
```

Equality can be written to compare object reference as well as bool and int types, and can appear as follows

```
var x = [1, 2]  
var y = x  
x == y //resolves to true  
or  
1 == 2  
or  
true != false
```

Print Statement

The Catscript print statement allows objects converted to string values to be outputted at evaluation or compiled runtime and can be written as follows

```
print(1)  
print([1, 2])  
print("Hello World")
```

If Statements

Catscript supports the basic if control flow statement. If statements are comprised an expression that evaluates to a bool and can be formatted to have an else if or an else as optional additions but can also be nested for desired output. An if statement can be written as follows


```

var x = 1
if (x == 1) {
    print("1")
} else if (x == 2) {
    print("2")
} else {
    if (x == 3) {
        print("3")
    } else {
        print("4")
    }
}

```

For loops

Catscript only supports a basic for loop control flow statement that uses a variable not predefined in another scope and an iterable list. The for loop will loop over every variable in the iterable list given, and can be written as follows

```

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

```

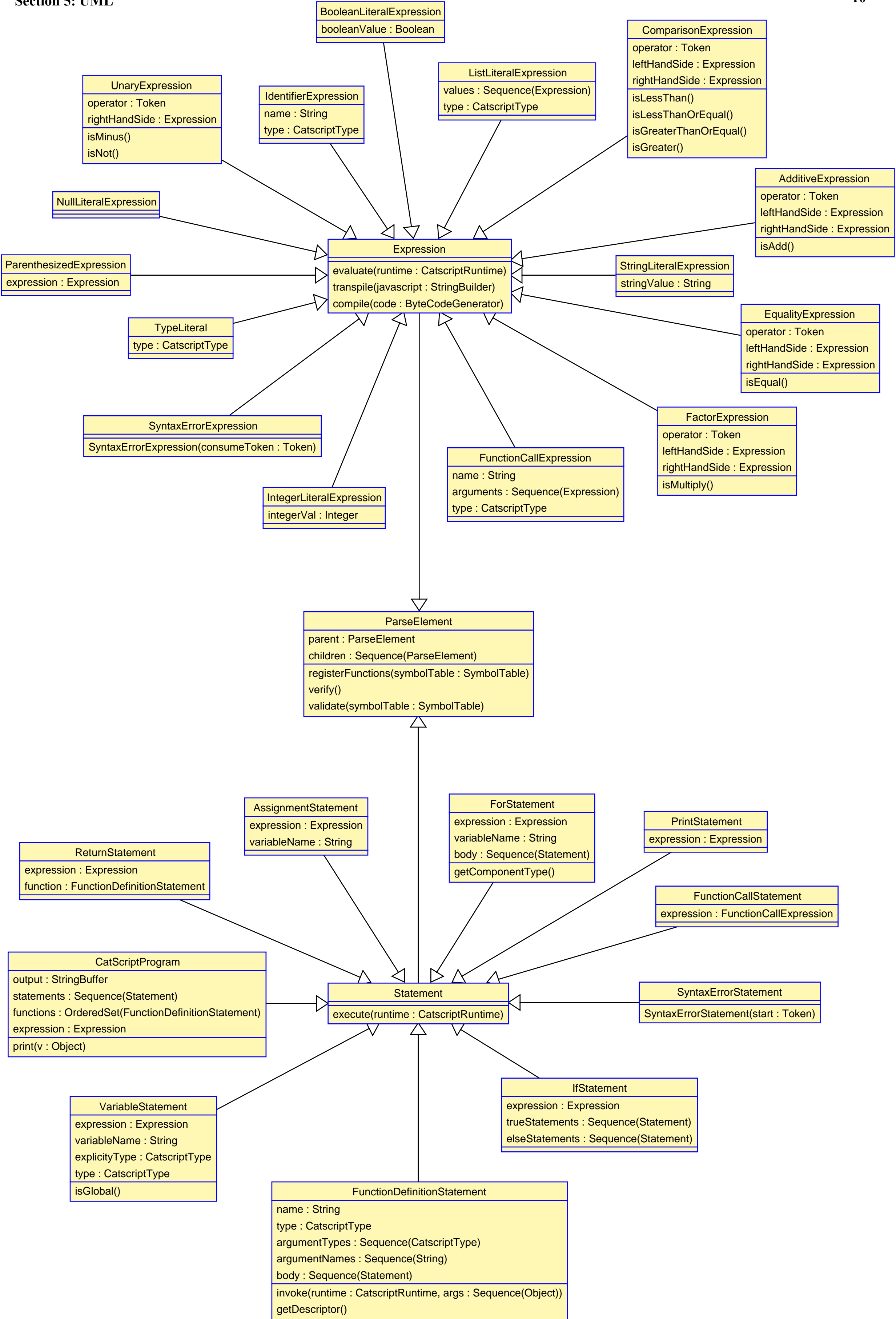
Functions

Functions in Catscript can be defined with a return type but will infer a return type if none is given defaulting to void, functions can be defined with any amount of parameters with an explicit type but can be inferred if omitted. Function identifiers cannot be used more than once and does not support method overloading. Function calls evaluate to a value or void given the provided return type. Function definition and call can be written as follows

```

function foo(x:int, y:int):int{
    return x + y
}
print(foo(2, 3))

```



Section 6: Design Trade-offs

Throughout this project there needed to be a design trade-off made when choosing an algorithm to create the parser. In this compiler, for the parser, I used recursive descent rather than a parser generator for a number of design reasons. First off, recursive descent more naturally goes with the recursive nature of context-free grammars. The trade-off is for educational purposes as a better and a more deep understanding of grammars can be obtained throughout the process of using recursive descent. Another reason for using recursive descent is the simplicity that comes with it as it more closely reflects grammars than a parser generator. While being more simple to understand and more informative than a parser generator the recursive descent algorithm requires more writing and work to create than using the parser generator. All in all, while a parser generator would have required less work overall it was a better trade-off to go with recursive descent in order to create a better understanding of grammars as well as parsers as a whole.

Section 7: Software development life cycle model

Throughout this capstone project the Test Driven Development model was used. The Test-Driven Development model is a development process in which users write tests that need to be passed by the code they are writing for a project. In other words, projects that employ this development method first write a test suite in code that will run tests on the actual project's code and make sure it meets the necessary requirements to make it work properly. Before starting this

project there was a test suite written for us, of which all relevant tests had to be passed in the end in order to ensure we created a fully functional compiler. Using this model helped immensely throughout the project because it kept our progress on track and organized. I saw this design model as being a very useful method for helping us really understand what we were doing and why it needed to be done.