

Catscript Compilers Project

Cameron Wilcox, Makayla Broyles

Gianforte School of Computing, Montana State University

CSCI 468: Compilers

Carson Gross

Spring 2023

I. Program

A zip file with the source code has been included with this document. See the directory /capstone/portfolio. The language spec is included here:

```
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 ,
'>' ]

```

II. Teamwork

For this project we were full time developers on the project, so we each wrote our own compiler, however, for the documentation and parts of testing, we assisted each other. We wrote three tests for each other, and wrote the technical documentation for Catscript for each other.

This can be seen later in the document. The tests that I wrote can be seen here:

```

@Test
void forLoopPrint() {
    assertEquals("1\n2\n3\n", executeProgram("for(x in [1, 2, 3]) { if(true){ print(x)} }"));
}

@Test
void ifStatementTest() {
    assertEquals("1\n", executeProgram("var x = 1\n if(x == 1){ print(x)}"));
}

@Test
void funcReturnTest() {
    assertEquals("10\n", executeProgram("var x = 10\n" + "function foo(y : int) : int {\n" +
        "    return y\n" +

```

```
    "}\n" +  
    "print( foo(x) )\n"));  
}
```

The tests that my partner wrote can be seen here:

```
@Test  
void test1() {  
    assertEquals("12\n", executeProgram(" var x = 11 if(x<10){ print(x) } else {  
print(x+1) }"));  
}  
  
@Test  
void test2() {  
    assertEquals("2\n3\n", executeProgram("for( x in [1, 2, 3] ) {\n if(x>=2){  
print(x)}}\n "));  
}  
  
@Test  
void test3() {  
    assertEquals("0\n", executeProgram("function foo() : int { " +  
        "var x = 9" +  
        "if(x>10) {return x} " +  
        "else { return 0}" +  
        "}" +  
        "print(foo())"));  
}
```

III. Design Pattern

The design pattern that we used for this project was the memoization pattern. You can see the implementation below:

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

We did this so that in the case that it becomes very expensive to new up a type, then every time there is a type that comes in that has already been seen, you can just look it up instead of creating a new one.

IV. Technical Documentation/Catscript Guide

Introduction

Catscript is a simple scripting language. Here is an example:

```
var x = "foo"  
print(x)
```

Features

For Loop Statement

The for loop statement is a feature to iterate over a range of values. Here is an example of a *For Loop Statement*:

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

If (Else) Statement

The if (else) statement is a feature used to execute only a certain chunk of code ‘if’ the conditional statement is true. Here is an example of an *If (Else) Statement*:

```
if(x > 10){ print(x) }
```

Else Statement

The else statement is a feature that is used with the ‘if statement’. It is used as the alternative—the if statement is false—to the ‘if’ conditional statement. Here is an example of an *If Else Statement*:

```
if(x > 10)  
{ print(x) }  
else { print( 10 ) }
```

Print Statement

The print statement is a feature used to print any parameters given to it. Here is an example of the *Print Statement*:

```
print(1)
```

Variable Assignment

The variable assignment feature is used to declare a variable in Catscript. You can declare a variable with or without any explicit type. Here is an example of a non-explicit type *Variable Assignment*:

```
var x = true
```

Here is an example of an explicit type *Variable Assignment*:

```
var x : bool = true
```

Function Declaration

The function declaration feature is used to define a function to be used within CatScript.

Here is an example of a *Function Declaration* with parameters:

```
function foo(a, b, c) {}
```

Here is an example of a *Function Declaration* without parameters:

```
function foo() {}
```

Function Call

The function call feature is used to call a declared function. Here is an example of a *Function Call*:

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

Return Statement

The return statement feature is used to return values from a function. Here is an example of a *Return Statement*:

```
function foo() : object { return true }
```

Variable Types

Catscript is statically typed and uses the following types.

List Literal

A list literal is a list of values with the type 'x'. Here is an example of a *List Literal*:

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

String

A string variable is a java-style string. Here is an example of a *String Variable*:

```
var x: string = "String"
```

Integer

An integer variable is a 32 bit integer. Here is an example of an *Integer Variable*:

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

Boolean

A boolean variable is a boolean value (true or false). Here is an example of a *Boolean Variable*:

```
var x : bool = true
```

Object

An object variable is a value of any type.

Operators

Arithmetic Operators

+	Additive Operator
-	Subtraction Operator
*	Multiplication Operator
/	Division Operator

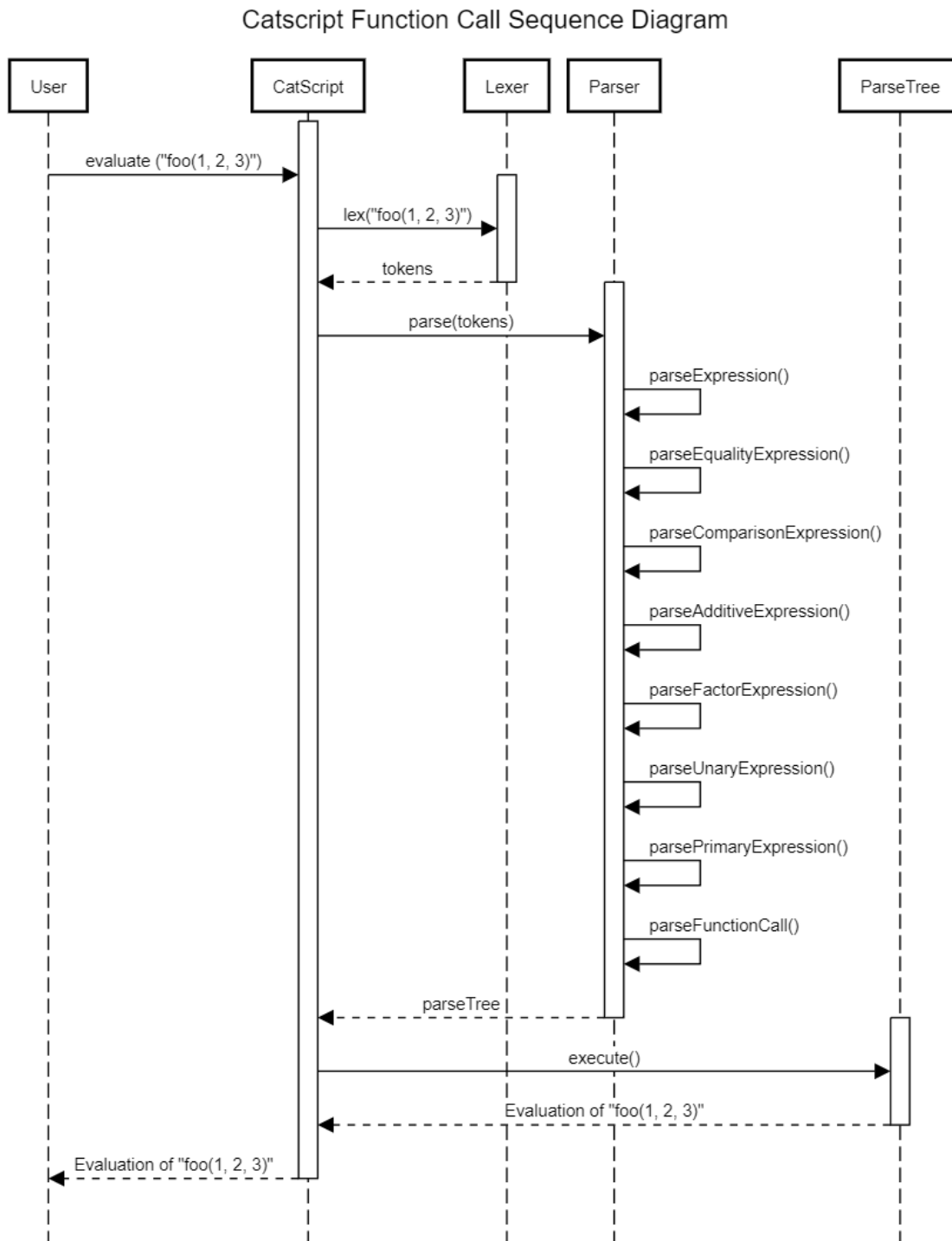
Unary Operators

+	Unary Plus Operator
-	Unary Minus Operator
++	Increment Operator
--	Decrement Operator
!	Complement Operator

Comparison Operators

>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

V. UML Diagram



For the UML Diagram, I did the parsing of a function call. It goes down the recursive descent tree before hitting `parseFunctionCall()` at which point it returns back up with the evaluation of the function.

VI. Design Trade-Offs

In this project, the main trade off that we made was that we wrote the parser by hand instead of using a tool like Lex or Yacc. We did this so that we could write a recursive descent parser that would both teach us much more about parsing, and so that the parser is optimized more for Catscript as a language.

A secondary trade off that we made in this project was the exclusion of the visitor pattern. We made this as we felt that the visitor pattern over complicated things greatly. However it is worth noting that with that trade off that we made, it does make things a bit more tightly coupled. We felt that this was a compromise we could make in order to make the compiler much more simple.

VII. Development Life Cycle

For this project we used Test Driven Development (TDD). This model was most helpful because we could write the different parts of the compiler as we went along and test each part individually and ensured that it was working every step of the way. We felt that this approach was far more helpful than detrimental and helped us to learn how to properly write a compiler the whole way through.