

COMPILERS CAPSTONE

CSCI 468

Spring 2023

Amanda Faulconer

Section 1: Program

All the partner tests that both Rory and Amanda wrote are in the PartnerTests.java file. Here is the absolute file path:

C:\Users\Amanda Faulconer\csci-468-spring2023-private\src\test\java\edu\montana\csci\csci468\parser\PartnerTests.java

Section 2: Teamwork

The capstone project was executed through a series of test-driven development sessions. Team member one (Amanda Faulconer) invested 90% of the time spent and team member two (Rory Myer) invested 10% of time spent into the completion of all the sets of tests. The final set of tests were developed by team member two, they were subsequently provided to team member one. Team member one executed the tests provided which completed the final round of testing.

Section 3: Design Pattern

The Memorization Pattern was the major design pattern that was implemented in the software development of the Catscript compiler. This pattern is used within the CatScriptType.java file to create list types in which each element can be stored within itself allowing for memory space to be freed.

```
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;
}
```

Section 4: Technical Documentation

Catscript Guide

Introduction

Catscript is a simple scripting language that compiles JVM Bytecode.

Features

Types

Catscript supports basic types for variables and is statically typed

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

- **Int**
A 32 bit integer
- **String**
A Java-style string
- **Bool**
A boolean value
- **List**
A list of values with type 'x'
- **Null**
The null type
- **Object**
A value of any type

Keywords

```
else, if, for, function, not, null, print, return, true, false, var
```

Expressions

Catscript uses an expression class inherited by all expression types.

- **Primary**

The default expression. Contains the following:

- Identifiers
Represent a user defined keyword or variable
- String literals
String of characters enclosed in quotation marks
- Boolean values
True or false values
- Null values
A null value
- List literals
A list of values, can be composed of the previously addressed expressions
- Function calls
Called at runtime, contains function parameters
- (a series of) expressions
Expressions of any type

- **Unary**

Catscript implementation of single argument expressions. Unary expressions in catscript use the 'not' symbol for boolean values and '-' for integers.

```
-x  
not false
```

- **Additive**

Catscript additive expressions implement addition and subtraction using the '+' and '-' operators, respectively.

```
3 + 7 // =10  
7 - 3 // = 4
```

- **Factor**

Catscript factor expressions implement multiplication and division using the '*' and '/' operators, respectively.

```
3*3 // =9  
3/3 // =1
```

- **Comparison**

Catscript comparison expressions implement less than, greater than, less than or equal to, and greater than or equal to, using the '<', '>', '<=', '>=' operators, respectively.

```
1 < 3 // true
```

- **Equality**

Catscript equality expressions implement equal to and not equal to, using the '==' and '!=' operators, respectively.

```
3 == 3    // true
3 != 3    // false
```

Statements

- **Print Statement**

Catscript print statements are implemented by using the 'print' keyword followed by an expression. The expression is evaluated and sent to the print command.

```
print(11)           // prints '11'
print("Hello World") // prints 'Hello World'
```

- **Variable Statement**

Catscript variable statements are used to declare and assign variables, implemented using the 'var' keyword.

```
var mascot : string = 'Champ'
var x : int = 11
```

- **Assignment Statement**

Catscript assignment statements assign values and expressions to variables using the equal ('=') symbol.

```
x = 11
x = x + 3
```

- **If Statement**

Catscript if statements are used for conditional branching. If the condition stated after the 'if' keyword is met, the code enclosed by the statement will be executed.

```
var x = 11
if (x == 3) {
    print(3)
} else {
    print(var)
}
```

- **For Statement**

Catscript for statements are used to initiate a loop on the condition following the 'for' keyword.

```
var numbers : list<int> = [1,2,3]
for (var i in numbers) {
    print(i)
}
```

- **Function Definition Statement**

Catscript function definition statements define functions used in a program. Indicated by the 'function' keyword, followed by an identifier, parameter list, and potentially a return statement.

```
function multiply(x: int, y: int) {
    return x*y
}
```

- **Function Call Statement**

Catscript function call statements execute functions previously defined if parameters are met accordingly.

```
multiply(3, 11);
```

- **Return Statement**

Catscript return statement uses the 'return' keyword to assign the return value to its function definition.

```
return x*y
```


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];

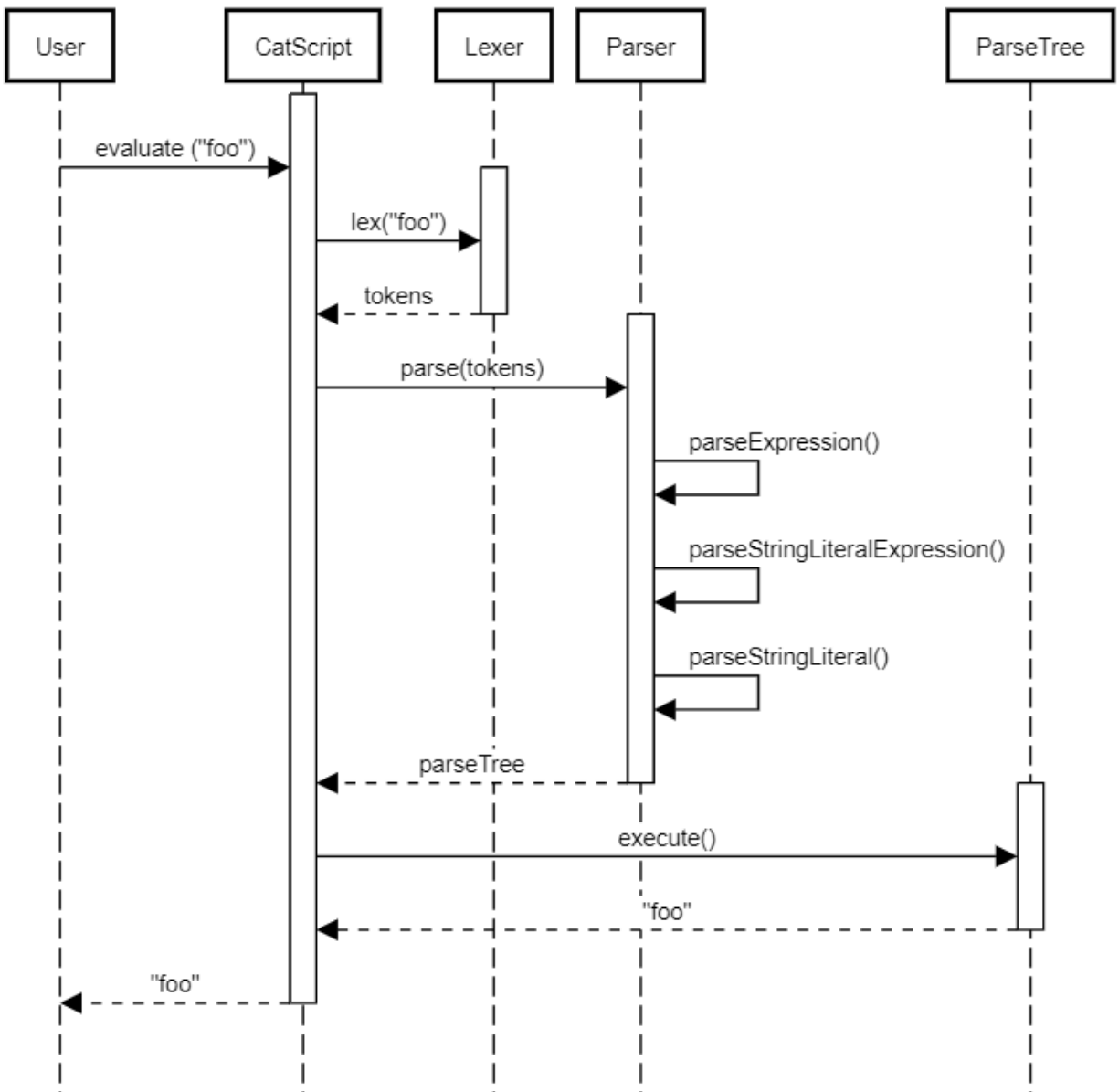
expression = equality_expression;

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


Section 5: UML

The following UML Diagram shows the overall structure of the Catscript Language.

Catscript Sequence Diagram



Section 6: Design Trade-offs

For the Catscript parser, a handwritten recursive descent algorithm was implemented. Recursive descent parsers are top-down parsers, which can be more time-consuming as all of the code must be handwritten, instead of using a parser generator. The programmer is required to create a tokenizer, parser, evaluator, bytecode generator, and JavaScript transpiler. This demands significant effort from the developer and is not very space efficient. However, by handwriting the parser, the programmer can gain a deeper understanding of all the components involved in building a parser. This, in turn, can make the parser faster as the parse tree is built from the top-down, starting with the non-terminals and relating them to a procedure, with no backtracking. In the end, the programmer can have an incredibly useful and fast data structure to build upon.

One major advantage of a recursive descent parser is that each non-terminal is related to a procedure. The objective of each procedure is to read a sequence of input characters or a string, which can be produced by the corresponding non-terminal, and return a token to the root of the parse tree for that non-terminal. For each production in the grammar, a method is created, named after it. For example, 'parseEquality()'. Within this method, the methods defined on the right-hand side of the production are called, matching strings as needed.

Section 7: Software Development Life Cycle Model

Test Driven Development (TDD) was utilized for this project. TDD is a software development process where test cases are created to validate and specify each function within the program. The tests are written before the code and will initially fail until the code is written to make them pass. This method involves repeatedly testing the software against all the test cases to track its development progress.

Initially, the method seemed overwhelming to us because of the number of failing tests. However, after the first checkpoint, we found it to be a preferred approach to software development. By having tests to work off of, we can isolate specific sections of the code that require improvement. If we had used traditional software development, we would have had to backtrack to locate the errors and written many lines of code.