

Catscript Compiler

CSCI 468

Capstone Project Portfolio

Rory Myer

Montana State University

Spring 2023

Section 1: Program

A zip file of the source code is included in this directory in source.zip.

Section 2: Teamwork

The capstone project was executed through a series of test driven development sessions. Team member one (Rory Myer) invested 90% of time spent and team member two (Amanda Faulconer) 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. Both team member's tests can be found in the CapstonePartnerTest.java file within the parser tests folder.

Section 3: Design Pattern

The memoization design pattern was used to complete the catscript compiler. A specific example of the use of memoization is seen in the `getListType` function of the `CatscriptType` file. The pattern is used to store the computation results in the `LIST_TYPES` hashmap in order to save the computation of a new list on each call to `getListType()`. Memoization was included as opposed to coding the new list directly each time for space efficiency.

```
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 Writing

Catscript Guide

Introduction

Catscript is a sophisticated yet simple, statically typed scripting language. An example of the Catscript language can be seen here with more example to follow in the featured section:

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

Features

Catscript Type System

Type Literals

Type Literals (Type Expressions) are simple pieces of data within the Catscript language. The data is stored in tokens and the type is recognized by the tokenizer. The parser will then store the data in a structure to evaluate and compile it at runtime. Type Literals may consist of INT, STRING, BOOLEAN, OBJECT, NULL, VOID, and LIST.

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

int

A 32-bit integer

string

A java-style string

bool

A Boolean value (true/false)

list

A list of values with the type 'x'.

```
list<type literal>
```

null

The null type

object

Any type of value

void

The type of no type (a function that does not return a value)

Catscript Keywords

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

Expressions

Primary Expressions

Primary Expressions are defined by eight expressions: Identifier, String Literal, Integer Literal, Boolean Literal, Null Literal, List Literal, Function Call Expression, and Parenthesized Expressions.

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" |  
"null" | list_literal | function_call | "(", expression, ")"
```

- **Identifiers** are expressions that represent a keyword or variable that is defined by the user.

```
x  
y  
z
```

- **String Literals** are expressions that represent a string of character values surrounded by quotation symbols.

```
"Hello, World"
```

- **Integer Literals** are expressions that represent integer values.

```
1  
2  
3
```

- **Boolean Literals** are expressions that represent true or false values.

```
true  
false
```

- **Null Literals** are expressions that represent null values.

```
null
```

- **List Literals** are expressions that represent a list of values. These can be Integer, String, Boolean or a combination of values.

```
["a", "b", "c"]  
[1, 2, 3]
```

- **Function Call Expression** contains information within the function parameters that is being called at runtime.

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

- **Parenthesized Expressions** are a set of expressions or values that are surrounded by parentheses. These expressions can be of any type.

```
1 + (1 + 1)
```

```
("foo" + "foo")
```

Additive Expressions

Additive Expressions are expressions that have two arguments separated by a plus or a minus symbol. The separated expressions may be Integer Literals, String Literals, or Parenthesized Expressions that contain Integer Literals or String Literals. The evaluation order is: LHS to RHS by precedence of the operator.

Examples:

```
( "+" | "-" )
```

```
"a" + "b"
```

```
2 - 1
```

Comparison Expressions

Comparison Expressions consist of two Integer Literal expressions that are separated by a less than, greater than, less than or equal to, greater than or equal to symbol.

Examples:

```
1 < 2
```

```
3 > 1
```

```
x <= y
```

```
y >= x
```

Equality Expressions

Equality Expressions are expressions that are separated by either a double equal (both sides are the same) or a bang equal symbol (both sides are not the same).

Examples:

```
True == True
```

```
True != False
```

Unary Expressions

Unary Expressions are expressions with a single argument. These expressions contain the 'not' symbol and the negative symbols. They may be applied to Boolean Expressions and Integer Literals.

Examples:

```
( "not" | "-" )
```

```
var x = -3
```

```
if(not false) {...}
```

Factor Expressions

Factor Expressions consist of two expressions that are separated by a multiplication (asterisk) or division (slash) symbol. The two expressions may be either Integer Literals or Parenthesized Expressions that contain Integer Literals.

Examples:

```
( "*" | "/" )
```

```
2 * 2
```

```
6 / 3
```

```
(4 * 2) / 2
```

Statements

For Statement (For Loops)

For Loops allow for iteration over lists while performing operations for each iteration. They will continue as long as there is a next element in the list. These statements are able to contain any other type of statement within it.

Example:

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

If Statement

If Statements handle conditional branching that restricts access to portions of the code until the conditional statement is true or met. They are typically followed by an Else Statement which will contain the false condition of the If Statement.

Example:

```
if ( int : x > int : y ) {  
    print ("x is greater than y") //true statements  
} else {  
    print ("x is less than y") //false statements  
}
```

Print Statement

The Print Statement recognizes the print command and will print the expression inside the parentheses to standard out. These statements can bring Integer and String values.

Examples:

```
print("Foo Bar")
```

Output: Foo Bar

```
print(1 + 1)
```

Output: 2

Variable Statement

Variable Statements are used to assign a value to a variable the user creates. These statements require the keyword 'var' so the parser knows to store the expression into the identifier that follows the keyword. Variables can be initialized explicitly or implicitly. To invoke a variable to be 'null', it must be done explicitly.

Examples:

```
var x : string = null
var x = 1
```

Assignment Statement

Assignment Statements assign an expression to an identifier by looking for the equal symbol. These are used when a variable is being assigned to a different value.

Examples:

```
x = 10
x = x + 10
```

Function Call Statement

Function Call Statements can be both an expression and/or a statement. Both the expression and statement require an identifier to start. The Function Call Expression will look for an identifier or function call and invokes the Function Definition Statement. The Function Call Statement will look for an assignment or a function call.

Example:

```
foo(10) //calling the function with the argument 10
function foo (i : int) {
    print(i)
}
```

Function Definition Statement

Function Definition Statements are the statements that define the functions that are being called within the program. They require the 'function' keyword to start, followed by an identifier, parameter list, an optional return statement, and a series of statements including a new return statement. The body of the Function Definition statement can contain any number and/or types of statements. All of which must be inside the function.

Example:

```
function x ( a : int, b: int, c : int) {  
    return (a * b * c)  
}
```

Return Statement

The return statement looks for the 'return' keyword, ends the function they are contained within, evaluates the expression, and assigns the return value to the function definition.

Examples:

```
return CatscriptType.INT  
return 1 + 1  
return null
```

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, '>']
```

Section 5: UML

Figure 1: Factor Expression

CatScript Factor Expression Sequence Diagram

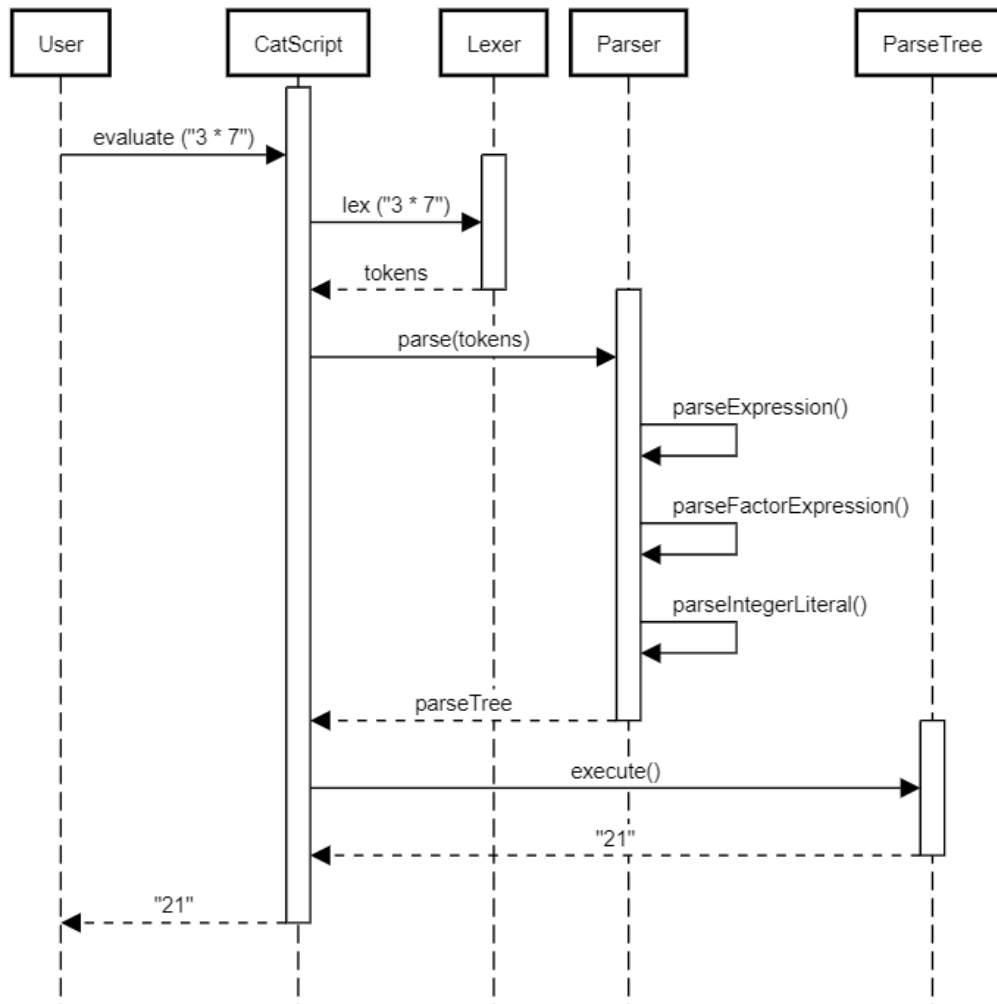
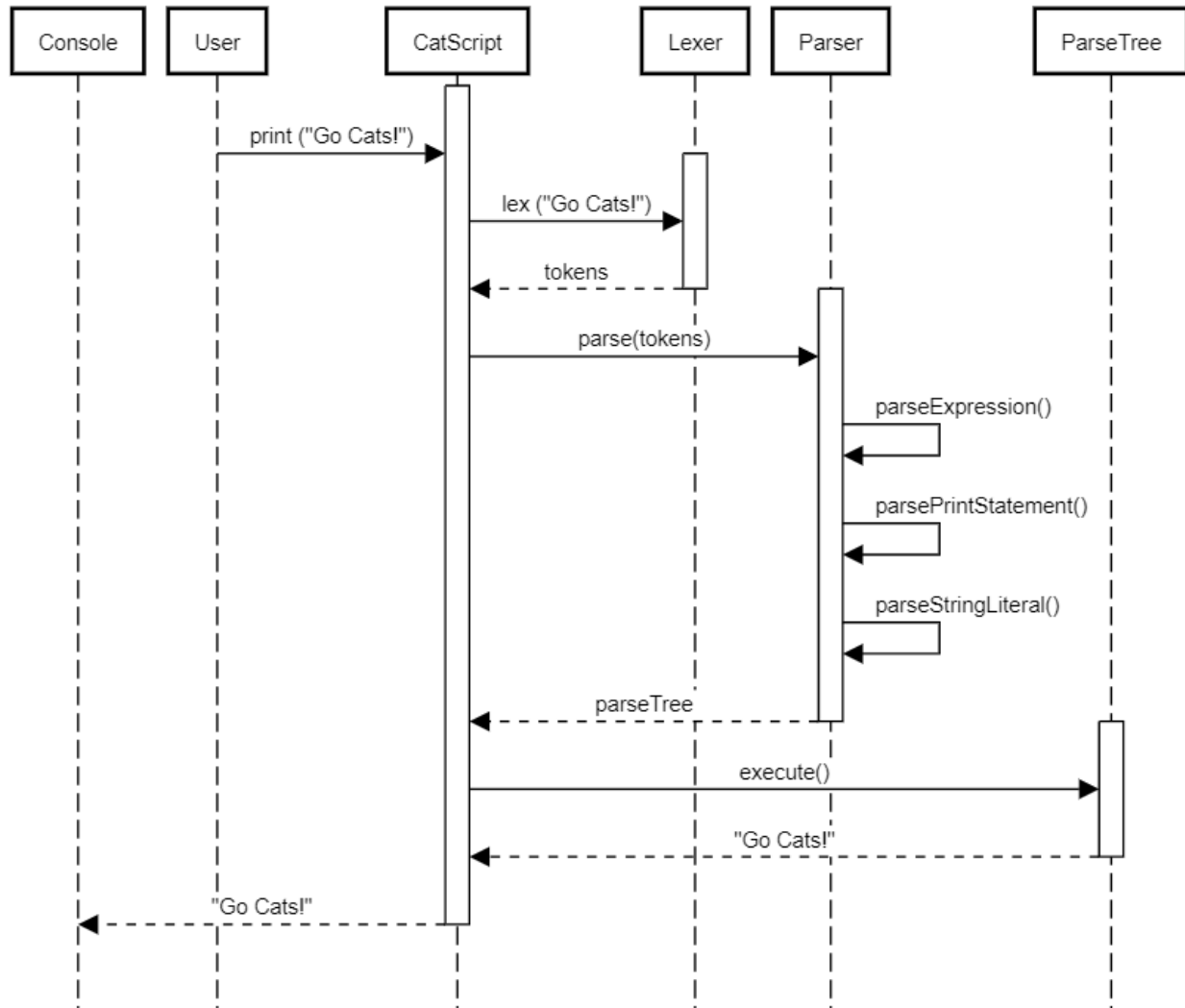


Figure 2: Print Statement

CatScript Print Statement Sequence Diagram



Section 6: Design Trade-offs

A handwritten recursive descent algorithm was used for the completion of the parser. The recursive descent algorithm was used in opposition to a parser generator, also used to develop parsers. Although either approach would result in a complete parser, the handwritten recursive descent parser initiated a more hands-on learning approach than the parser generator would have.

Coding all aspects of the language by hand in the order in which they were coded initiated an understanding from the barebones of the JVM, up. Starting with tokenization, also referred to as lexical analysis, strings were split into individual tokens to be fed to the parser in the following steps. Expression parsing came next, at which point the strings from which the tokens were derived were parsed. Statement parsing and evaluation built on expression parsing, at which point bytecode followed.

Had a parser generator been used, the previously examined steps would not have been understood in the depth the handwritten method allowed for. The purpose of the capstone course is for students to generate an in-depth knowledge of how all components of a parser work together to parse a language. The goal was achieved due to the nature of the recursive descent compiler.

Section 7: Software Development Life Cycle Model

Test Driven Development (TDD) was used to develop the catscript compiler. A test suite was provided and used incrementally throughout the semester to complete the parser. Tests would fail until corresponding code was corrected; the tests clearly outlined what was expected of the parser in order for them to pass. The use of the tests and debugger side by side paved the way for a functionally complete catscript compiler.

The model helped drastically, it was a hands on approach that allowed for a deep understanding of how each part works individually. The test sets were sequentially deployed in a logical manner that allowed for the best understanding of how all these individual components work together under the hood.