

CSCI 468 Portfolio

Montana State University

Abigail Quinn | Jack Brown

Spring 2023

Section 1 : Program

Source code can be found at

<https://github.com/abbykq/csci-468-spring2023-private/tree/main/src>

Section 2: Teamwork

In this project, the work was divided into a 95%-5% split, with the first team member (Abigail Quinn) taking on all of the automated testing responsibilities. This team member completed the testing process by completing the four checkpoints: tokenization, parsing, evaluation, and bytecode generation. The second team member (Jack Brown) contributed to the test-driven development cycle by writing an additional three tests. Each partner wrote the Technical Documentation for the other partner.

Section 3: Design Pattern

The **Memoization Pattern** used in the CatscriptType.java class optimizes the storage and retrieval of previously computed results. By mapping types to their corresponding lists in a HashMap, the getType() method avoids redundant computation, resulting in improved performance. This design pattern is especially useful in scenarios with expensive computations or object creation.

```
36 // TODO memoize this call
37 2 usages
38 private static final Map<CatscriptType, CatscriptType> LIST_TYPES = new HashMap<>();
39 6 usages Carson Gross *
40 @ public static CatscriptType getListType(CatscriptType type) {
41     CatscriptType listType = LIST_TYPES.get(type) ;
42     if(listType == null){
43         listType = new ListType(type);
44         LIST_TYPES.put(type,listType);
45     }
46     return listType;
47 }
```

Section 4: Catscript Technical Documentation

(Jack Brown)

Introduction

Catscript is a statically typed simple expression built using recursive descent. The finished project results in a functioning compiler. Examples of Catscript code can be seen below.

Statements

For Statement

Uses an identifier(variable name) to iterate over an expression. A statement is executed for each iteration.

- Grammar: 'for', '(', IDENTIFIER, 'in', expression ')', '{', { statement }, '}'

- ```
for (i in [2, 4, 6]){
 print(i)
}
```

**Output:** 2, 4, 6

#### If Statement

Executes a statement if an expression(condition) returns true, otherwise the program executes another statement.

- Grammar: 'if', '(', expression, ')', '{', { statement }, '}' [ 'else', ( if\_statement | '{', { statement }, '}' ) ]

- ```
if (20 > 10) {  
    print("true statement")  
}  
else {  
    print("false statement")  
}
```

Output: "true statement"

Print Statement

Prints an expression in the Catscript Console

- Grammar: 'print', '(', expression, ')'
- `print("Test")`
Output: "Test"

Variable Statement:

This statement initialized a variable with a given name (identifier). This variable can either be initialized with an implicit type or a given type.

- Grammar: 'var', IDENTIFIER, [':', type_expression,] '=', expression;
- `var x : int = 10`

Assignment Statement

Assigns an expression (a value) to a variable. This happens after the variable has already been declared.

- Grammar: IDENTIFIER, '=', expression;
- `x = 10`

Function Call Statement

Calls a function using its identifier (method name) and a list of arguments.

- Grammar: IDENTIFIER, '(', argument_list , ')' *where* argument_list = [expression , { ',', expression }]
- `foo(x)`

Function Definition Statement

This statement creates/declares a function. It requires the 'function' keyword in order to start the statement, a list of parameters (0 or more) and a starting and closing bracket.

- Grammar: 'function', IDENTIFIER, '(', parameter_list, ')' + [':' + type_expression], '{', { function_body_statement }, '}'
- `Function foo(val) : object {}`

Expressions

Primary Expression

Primary expressions are the baseline types which are present in every language. The main eight in Catscript are IDENTIFIER, STRING, INTEGER, true, false, null, list literal, and function call. IDENTIFIER is a name which is associated with a value, often called a variable. A string is a zero or more characters combined together. An integer is a whole number. “True” and “false” are keywords which always contain the value true or false. “Null” is a keyword with an associated value of nothing. Its value is that it's empty. List literal is a list of any type of primary expression. Function call is a call to a function that already exists. This takes in a list of parameters to send to the function. Examples of each are shown below.

- Grammar: IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" | list_literal | function_call | "(", expression, ")"
- `i, j, testVar : IDENTIFIER expression`
- `"Test", "MSU", "16jr4" : STRING expression`
- `10, 20, -10 : INTEGER expression`
- `bool = true : "true" expression`
- `bool = false : "false" expression`
- `x = null : "null" expression`
- `[10, 20, 30], ["a", "b", "c"] : list_literal expression`
- `foo(x) : function_call expression`

Unary Expression

Unary expressions are a type of expression with one of two symbols attached to it: a not or a negative sign. These expressions can only be an integer literal or a boolean because those are the only two primary expressions which have two states of positive/negative or true/false.

- Grammar: ("not" | "-") unary_expression | primary_expression;
- `not True : returns False`

- `-20 : returns -20`

Factor Expression

Multiplies or divides two unary expressions with one another.

- Grammar: `unary_expression { ("/" | "*") unary_expression };`
- `20/10 : returns 2`
- `20 * 10 : returns 200`

Additive Expression

Adds or subtracts two expressions with one another.

- Grammar: `factor_expression { ("+" | "-") factor_expression };`
- `10 + 20 : returns 30`
- `10 - 20 : returns -10`

Comparison Expression:

Check if two expressions are less than, less than or equal to, greater than, or greater than or equal to one another.

- Grammar: `additive_expression { (">" | ">=" | "<" | "<=") additive_expression };`
- `10 > 20 : returns False`
- `10 >= 20 : returns False`
- `10 < 20 : returns True`
- `10 <= 20 : returns True`

Equality Expression

Checks if two comparison expressions are equal or not equal. Returns true or false depending on operand.

- Grammar: `comparison_expression { ("!=" | "==") comparison_expression};`
- `10 != 20 : returns True`
- `10 == 20 : returns False`

Types

int - a 32 bit integer

string - a java-style string

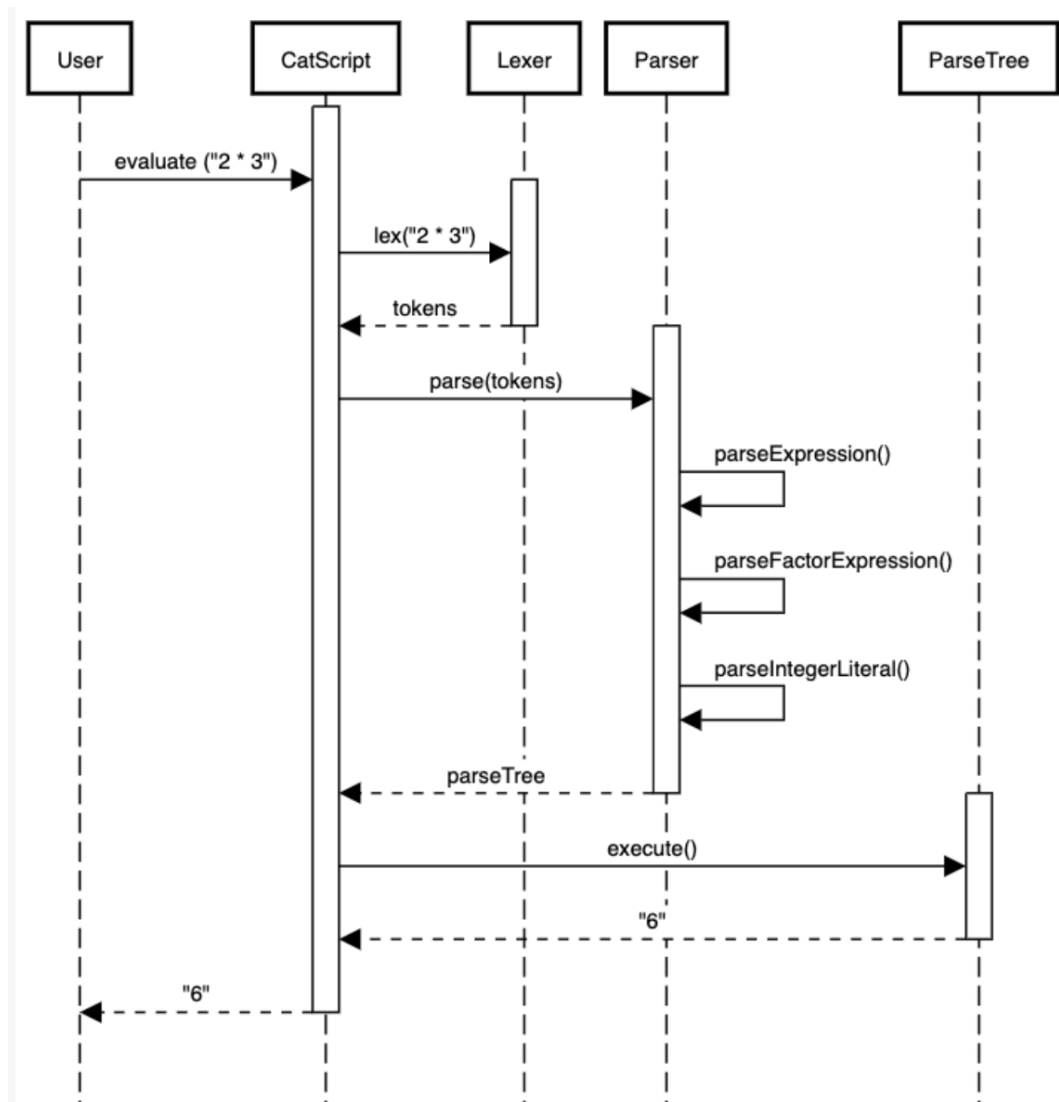
bool - a boolean value

list - a list of value with the type 'x'

null - the null type

object - any type of value

Section 5: UML



This sequence diagram shows the process of Catscript computing a multiplication expression. The user inputs the expression, then catscript tokenizes it, then the tokens are parsed, going through `parseExpression`, `parseFactorExpression`, and `parseIntegerLiteral`. This returns a parse tree that is then executed to return the correct total.

Section 6: Design trade-offs

The design tradeoff within this project was the decision to implement a handwritten recursive descent was used instead of a parser generator for developing Catscript parser. Although using a generator would be easier and take less time, writing the recursive descent parser by hand makes it easier to read, understand and modify.

Section 7: Software development life cycle model

Test Driven Development was the life cycle development model for this project. The four checkpoints were broken down into the following: tokenization, parsing, evaluation, and bytecode generation. Each checkpoint had several tests that needed to pass to create the complete Catscript compiler. The checkpoints relied on the previous one to work completely which helped with detection and resolution of any problems that could have potentially escalated into larger problems later.