

**COMPILERS - CSCI 468
SPRING 2023**

**JUSTIN KERR
WILLIAM WOOD**

A capstone submitted
in partial fulfilment of the requirements for the degree of

BACHELOR OF SCIENCE

in

COMPUTER SCIENCE

Gianforte School of Computing
Montana State University
BOZEMAN, MONTANA, UNITED STATES

Table of Contents

Section 1: Program	3
Section 2: Teamwork	4
Section 3: Design Patterns	6
Section 4: Technical Writing	7
Section 5: UML	14
Section 6: Design Trade-offs	15
Section 7: Software Development Life Cycle Model	16

Section 1: Program

The capstone project was to create a compiler for a statically typed functional programming language called Catscript. The compiler tokenizes a string of text, then parses the tokens as instructions which the computer can understand. Finally, the parsed instructions are translated into bytecode and then executed.

Source code can be found on <https://github.com/BillyCWood/csci-468-spring2023-private> in the capstone/portfolio folder.

Section 2: Teamwork

There were two roles in this project: a primary engineer and a documentation-and-testing engineer. Team member 1 was the documentation-and-testing engineer, and team member 2 was the primary engineer.

The role of team member 1 was document all the features that are present within the grammar of the Catscript programming language, i.e., how to use for loops, how to declare a variable, what are the data types, how are functions defined and called, etc. Team member 1 was also responsible for creating tests to validate the functionality of Catscript. Such tests included proper parsing of multi-dimensional lists, proper function calls with complex arguments such as lists or other functions, and testing to ensure proper arithmetic was taking place which included, but was not limited to addition and subtraction, multiplication and division, and exponents. The language documentation and unit tests were given to team member 2 after they had been developed.

The role of team member 2 was to develop the Catscript compiler to have the appropriate grammar as prescribed in the documentation, and to pass every unit test. This was achieved by first implementing a tokenizer, which would go through each word or character and assign a token type. Next all tokens would be parsed and determined to be either an expression or statement while also evaluating to having the proper functionality. Type checking and error handling was also implemented. Lastly, all expressions and statements would be translated into bytecode so that it would be able to be compiled and run properly on the Java Virtual Machine (JVM). The overarching algorithm or paradigm used in the implementation of the language was the recursive descent parsing algorithm, a top-down parser.

Both team members worked diligently on the Catscript project and were able to guide one another whenever needed. The amount of time each team member spent on the Catscript project was very similar, with percentage of time spent working from team member 1 coming to roughly 45%, and the amount of time spent working from team member 2 coming to roughly 55%.

Section 3: Design Patterns

One design pattern which was used in this project is the memoization pattern. This pattern can be found in the CatscriptType.java file at lines 38 through 46 at the method named getListType(). The memoization pattern is used to “remember” or store results corresponding to some set of specific inputs. Future method calls are then able to return the remembered result rather than performing repeated recalculations to return the result. This pattern optimizes the method’s speed by trading time cost for space cost. Time spent is reduced, but space used has increased. This pattern is specifically being used within the Catscript project to optimize type access. getListType() takes in one parameter “type.” The parameter “type” is then used against a hashmap to see if it has been stored. If it has been stored, then we return that result. If it is not stored within the hashmap, then we create a new ListType object and store type and ListType as the key-value pair in the hashmap so that future calls will have that result stored. This design pattern was necessary to use because although more space is being used by storing data within a hashmap, we perform less computations than the alternative, which is to create a new ListType object with every method call and return the newly created object.

Section 4: Technical Writing

Catscript Guide

Catscript is a simple scripting language. This guide aims to explain features in Catscript.

Catscript Types

Catscript supports the following types:

1. `int`: A 32-bit integer (whole number)
2. `string`: A Java-style string (text)
3. `bool`: A boolean value (true or false)
4. `list`: A list of values with a specified type
5. `null`: Represents the null type (an absence of value)
6. `object`: Can represent any type of value

Variables and Assignments

In Catscript, you can declare a variable using the `var` keyword, followed by the variable name, an optional type annotation, and an initial value. You can also assign new values to variables throughout the program using the `=` operator.

```
var variableName: type = initialValue;
variableName = newValue;
```

For example, declaring an integer variable and assigning a new value:

```
var myNumber: int = 10;
myNumber = 20; // myNumber is now 20
```

Catscript Features

Arithmetic

Catscript supports standard arithmetic operations such as addition, subtraction, multiplication, and division. These operations can be performed on expressions within the language.

For example, if you have two integers `a` and `b`, you can perform arithmetic operations like:

```
a + b // addition
a - b // subtraction
a * b // multiplication
a / b // division
```

Loops

Loops are used to repeat a set of instructions multiple times. Catscript includes `for` loops for iterating through a range of values or elements in a list. The syntax is as follows:


```
for (IDENTIFIER in expression) {  
    // statements  
}
```

For example, iterating through a list of integers:

```
var myList = [1, 2, 3, 4, 5];  
for (i in myList) {  
    print(i); // this will print each element in myList  
}
```

If Statements

Conditional statements are used to execute instructions based on whether a condition is met. In Catscript, you can use `if`, `else`, and optional `else if` statements

to create conditional statements. The syntax is as follows:

```
if (expression) {  
    // statements  
} else if (expression) {  
    // statements  
} else {  
    // statements  
}
```

For example, checking if a number is positive or negative:

```
var number = -5;  
if (number > 0) {  
    print("Positive");  
} else if (number < 0) {  
    print("Negative");  
} else {  
    print("Zero");  
}
```

Boolean Expressions

Catscript's boolean expressions can be categorized into three types:

1. Boolean Literals
2. Comparison Operators
3. Equality Operators

Boolean Literals

Catscript recognizes `true` and `false` as boolean literal values.

Comparison Operators

Catscript supports four comparison operators:

1. Greater than (`>`)
2. Less than (`<`)
3. Greater than or equal to (`>=`)
4. Less than or equal to (`<=`)

Examples of comparison expressions:

```
1 < 2 // evaluates to true
2 >= 2 // evaluates to true
2 > 2 // evaluates to false
2 > 3 // evaluates to false
```

Equality Operators

To check for equality in Catscript, use two equal signs (`==`).

Examples of equality expressions:

```
2 == 2 // evaluates to true
2 == 3 // evaluates to false
```

Negation Operators

Catscript provides two negation operators:

1. The keyword `not`
2. The exclamation mark (`!`)

`not` is used with boolean literals:

```
not true // evaluates to false
not false // evaluates to true
not can be used recursively:
```

```
not not true // evaluates to true
not not false // evaluates to false
```

`!` is used to negate equality expressions and to check for inequality:

```
2 != 2 // evaluates to false
2 != 1 // evaluates to true
```

Boolean Variables

Variables can be assigned with boolean expressions. The variable's value is the result of the expression:

```
var boolResult = 2 > 1 // the value of 'boolResult' is true
boolResult = 2 != 2 // the value of 'boolResult' is false
```

String

Strings in Catscript have the following properties:

1. Strings are immutable.
2. Strings cannot be indexed or sliced.

String Concatenation

Strings can be concatenated using the addition (+) operator:

```
var hello = "Hello"
var world = "World"
var greeting = hello + " " + world // "Hello World"
```

Strings can be concatenated with any data type to create a new string:

```
1 + "a" // "1a"
"a" + 1 // "a1"
null + "a" // "nulla"
"a" + null // "anull"
```

Functions

Functions are reusable pieces of code that can be called with specific inputs (called arguments), perform a series of instructions, and return a result. Catscript supports function declaration and calling. Functions can be defined with a specific return type or no return type at all. Function calls can be used as expressions or as standalone statements:

```
function functionName(parameterList): returnType {
    // function body
}
```

```
functionCall(arg1, arg2, ...);
```

For example, defining a function that adds two numbers and returns the result:

```
function add(a: int, b: int): int {
    return a + b;
}

var result = add(3, 5); // result is 8
```

Functions can also have no return type, in which case they can perform actions but do not return any value. For example, a function that prints the sum of two numbers:

```
function printSum(a: int, b: int) {  
    var sum = a + b;  
    print(sum);  
}
```

```
printSum(3, 5); // prints "8"
```

Comments

Catscript supports two types of comments:

1. **Single-line comments:** These comments span only one line and are denoted by double forward slashes (`//`). The compiler will ignore everything following the double slashes until the end of the line.

```
// This is a single-line comment in Catscript  
var number: int = 42 // This comment is inline with code
```

2. **Multi-line comments:** These comments span multiple lines and are denoted by a forward slash and an asterisk (`/*`) at the beginning and an asterisk and a forward slash (`*/`) at the end. Everything between the opening and closing comment markers will be ignored by the compiler.

```
/*  
    This is a multi-line comment in Catscript.  
    It can span multiple lines, making it suitable  
    for longer explanations or documentation.  
*/  
var message: string = "Hello, World!"
```

Unit Tests

```
package edu.montana.csci.csci468.parser;
import edu.montana.csci.csci468.CatscriptTestBase;
import edu.montana.csci.csci468.parser.expressions.*;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CatscriptNewTests extends CatscriptTestBase {

    @Test
    public void parseComplexExpression() {
        Expression expr = parseExpression("1 + 2 * 3 - (4 / 2) == 5", false);
        assertTrue(expr instanceof EqualityExpression);
        assertTrue(((EqualityExpression) expr).getLeftHandSide() instanceof AdditiveExpression);
        assertTrue(((EqualityExpression) expr).getRightHandSide() instanceof
IntegerLiteralExpression);
    }

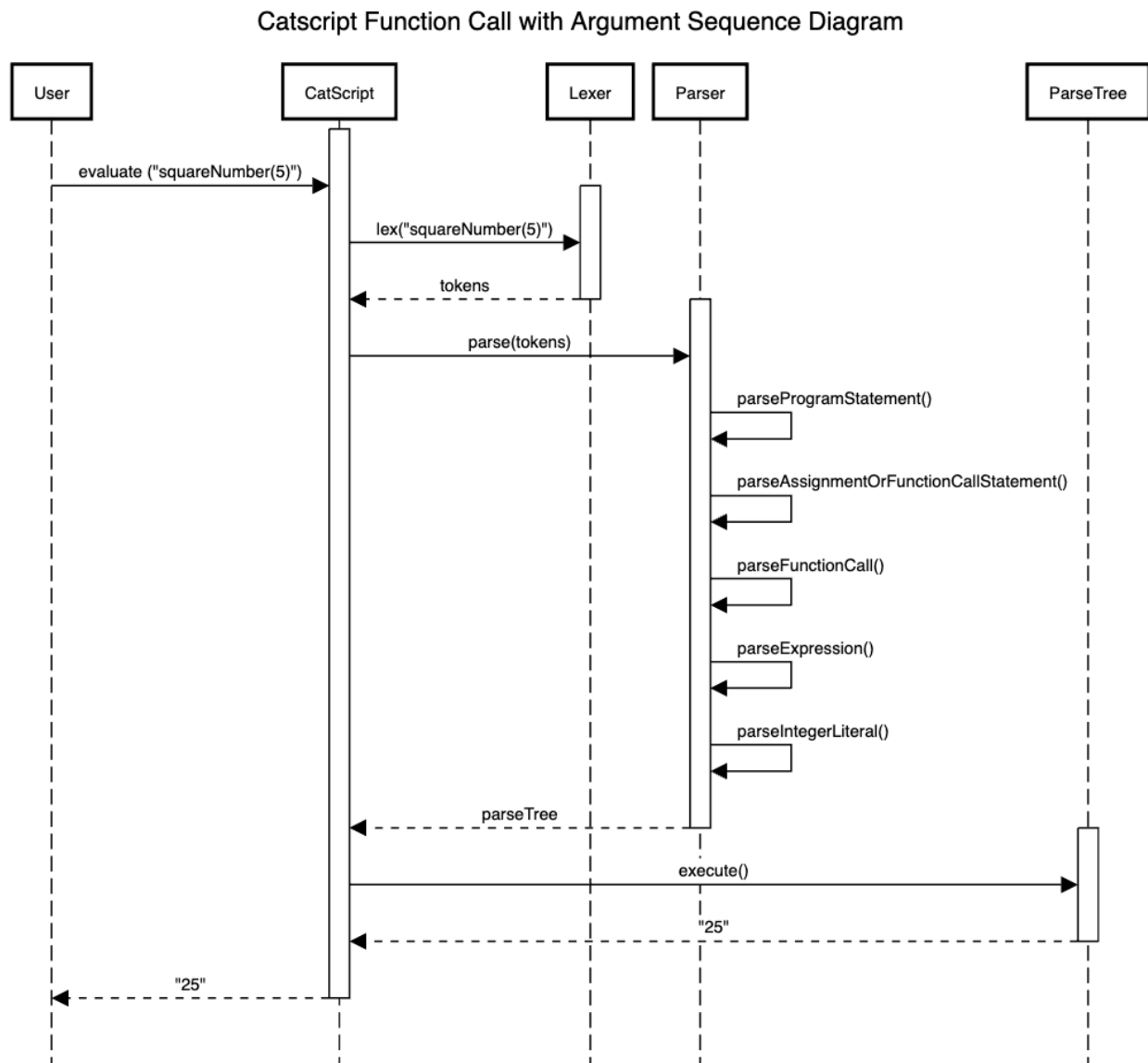
    @Test
    public void parseFunctionCallWithComplexArguments() {
        FunctionCallExpression expr = parseExpression("foo(1, bar(2, 3), [4, 5])", false);
        assertEquals("foo", expr.getName());
        assertEquals(3, expr.getArguments().size());
        assertTrue(expr.getArguments().get(0) instanceof IntegerLiteralExpression);
        assertTrue(expr.getArguments().get(1) instanceof FunctionCallExpression);
        assertTrue(expr.getArguments().get(2) instanceof ListLiteralExpression);

        FunctionCallExpression nestedFunctionCall = (FunctionCallExpression)
expr.getArguments().get(1);
        assertEquals("bar", nestedFunctionCall.getName());
        assertEquals(2, nestedFunctionCall.getArguments().size());
    }

    @Test
    public void parseNestedListLiteralExpression() {
        ListLiteralExpression expr = parseExpression("[1, [2, 3], 4]", false);
        assertEquals(3, expr.getValues().size());
        assertTrue(expr.getValues().get(0) instanceof IntegerLiteralExpression);
        assertTrue(expr.getValues().get(1) instanceof ListLiteralExpression);
        assertTrue(expr.getValues().get(2) instanceof IntegerLiteralExpression);
        ListLiteralExpression nestedList = (ListLiteralExpression) expr.getValues().get(1);
        assertEquals(2, nestedList.getValues().size());
    }
}
```

Section 5: UML

The following UML sequence diagram shows how Catscript executes a function call with arguments. First, the user calls the function “squareNumber(5)” to square whatever number it is given. Catscript then passes the call to the lexer, splitting the call up into tokens and passes them back to Catscript, which then gives the tokens to the parser to make sense of the tokens. The tokens are parsed recursively. After creating a parse tree, Catscript then executes the function, getting the number 25 and returning it to the user.



Section 6: Design Trade-offs

This project was designed using the recursive descent parser rather than using a parser generator. A major reason with going with the recursive descent route was readability. A hand-made recursive descent parser is unarguably more readable than code generated from a parser generator. It is easy to look at the code and read what a section might be doing because of meaningful and appropriate names for variables and methods. The improved readability also makes debugging the parser easier. The overall number of lines of code is also greatly reduced. The recursive decent tokenizer was built with roughly 265 lines of code, whereas the lines of code for the tokenizer produced by a parser generator is about six times that, which can be around 1200 lines of code.

The recursive descent parser gives more flexibility and freedom than parser generators. The parser generators will produce parse elements which the developer has no control over. But with recursive descent there is full control over the natural structure of parse elements, and methods such as evaluate or execute can be added to them. This also allows us to create a parser in just the language alone without the need of external parsing tools and any additional packages. Recursive descent design allows the developer to do as much as, or even more than, a parser generator with less.

Section 7: Software Development Life Cycle Model

The model used for this project was Test Driven Development (TDD). This development process relies on the requirements of the software being converted into test cases. These test cases are created before the software has been developed. The purpose of this approach was to track the development of the software to ensure that all the required functionality was being achieved by repeatedly running the software against all the test cases. The tests are meant to specify and validate what the code will do. Passing test cases indicate that the software has satisfied a requirement and shows how well developed the software as a whole product is. Failing test cases show where the software needs to be developed as requirements have not been met.

This form of development helped the team in a few ways. First, TDD, along with UML design, helped shape the design of the software. By focusing on the development of test cases before the development of software, the team of engineers were able to imagine the functionality used by the clients. The focal point became the interface rather than the implementation. This resulted in a clearer vision as to how the software should be, and how the implementation would work towards the desired interface. TDD helped the team during the development phase by taking small steps towards reaching the end goal. The engineers were able to focus on whatever task was at hand because the first goal in TDD is to get a passing test. This required the team to view the software as small units which could be written and tested independently from each other. An approach such as this helps keep code as clean, as clear and as simple as possible because only what is functionally required is implemented. Other functionality is not added until it is deemed necessary. This simple approach was very useful in debugging a program, allowing

the engineers to either revert to a previous version of the software in which a test or multiple tests which began to fail were now passing once more, or to step through the control flow of the software to discover what problem kept the team from passing a test. There is great confidence in the implementation among the team because all the code that has been written is covered by at least one test case.

However, TDD does come with its drawbacks, and hinderances were experienced during development. The main hinderance was not running tests frequently. In getting one test to pass and not running all previous tests against the code, there were instances in which a test which had been passing was now failing, and because further development had occurred since that test was last run and passed, it was difficult and tedious to find out where the error arose and how to remedy it. This was caused by a false sense of security in the abundance of passing tests which led to not giving a thought about having to rerun previously passing tests.