

CSCI 468 Capstone Documentation

Compilers

Spring 2023

Braeden Hunt

Rory Donley-Lovato

CSCI 468 - Compilers - Spring 2023

Braeden Hunt & Rory Donley-Lovato

Section 1: Program

The source code can be found in [csci-468-spring2023-private/capstone/portfolio/source.zip](https://github.com/csci-468-spring2023-private/capstone/portfolio/source.zip).

Section 2: Teamwork

For the capstone project, the work was divided into two parts among the two engineers. Team member 1 was responsible for developing and implementing the compiler. He was also responsible for managing the GitHub repository, keeping it up-to-date with commits from upstream, and handling any conflicts. Team member 2 was responsible for project documentation and creating additional high-level tests for the compiler.

Communication was done primarily through Discord and in-person conversations. Total Time Estimation: 100 hours Team Member 1: Contributions: Developer and DevOps Engineer. Estimated Work: 80 hours, 80% of total time. Team Member 2: Contributions: Technical Documentation and Quality Assurance. Estimated Work: 20 hours, 20% of total time.

Section 3: Design pattern

A primary driver when creating a compiler is the efficiency and speed of the compiler running. To increase this and save resources, we followed the Memoization design pattern. This design pattern allows us to reduce expensive operations by avoiding repeating the same function calls with the same arguments. We do this specifically in the `CatscriptType` constructor and `getListType()` method. These can be found in the file `src/main/java/edu/montana/csci/csci468/parser/CatscriptType.java`. We store a static map of strings and `CatscriptTypes`. Every time a new `CatscriptType` is created, it is stored in the map. This allows us to see if we have already created this type before and reuse it if we have before, as all `CatscriptTypes` are automatically added to the map. We implement this in our `getListType` function by first checking if the list type is in the `StringTypeMap`, returning the existing value if it does, and creating a new one if it doesn't. As we automatically add the types into the `StringTypeMap` on construction, the new list types get populated as well. The code to do this is part of sections 1 and 2.

1.

```
public static Map<String, CatscriptType> StringTypeMap;
```

2.

```
public CatscriptType(String name, Class javaClass) {
    this.name = name;
    this.javaClass = javaClass;
}
```

3.

```

    if (StringTypeMap == null) {
        StringTypeMap = new HashMap<>();
    }
    StringTypeMap.put(name, this);
}

public static CatscriptType getListType(CatscriptType type) {
    CatscriptType listType = StringTypeMap.get("list<" + type.toString() + ">");

    if (listType != null) {
        return listType;
    }
}

```

4.

```

    return new ListType(type);
}

```

Section 4: Technical Report

Introduction

This program is a compiler written in Java and creates a new language called Catscript. Catscript is a statically typed functional programming language. It possesses many of the features present in Java, such as for loops, if statements, and return statements.

Features

For Loop

Although built in Java, which is known for its three part for loop declaration, the Catscript for loop is quite simple. The grammar for this loop is below.

```

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
                '{', { statement }, '}'

```

This for loop's declaration only has two parts: an identifier variable and an expression to iterate through, most commonly a list. Every time the loop iterates through the guiding expression, the statements inside will be executed, using the argument variable if called, x in the example below.

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

If Statement

The Catscript If statement is straightforward like If statements in other languages. The controlling expression requires an expression that can return a boolean value. The grammar for the If statement is shown below.

```
if_statement = 'if', '(', expression, ')', '{',
               { statement },
               '}'
               [ 'else', ( if_statement | '{', { statement }, '}' ) ]
```

If the controlling expression returns true, the statement inside will execute. Else statements are also usable in Catscript. If the controlling expression returns false, the else statement will execute if present.

In the example below, the controlling expression returns true because the value of y is less than 3. If the variable y was greater than or equal 3, the else statement would execute instead and return false.

```
var y = 0
if(y < 3){
  return true
} else {
  return false
}
```

Variable Statement

Variable statements have three required parts, with an optional type expression. These required pieces are the var keyword, an identifier or variable name, and the expression that being assigned as the value. The grammar for this is shown below.

```
variable_statement = 'var', IDENTIFIER,
                    [ ':', type_expression, ] '=', expression
```

Variables can be set to any of six types: int, bool, object, string, list, and null. If the expression does not match the type set by the type expression, incompatible type errors will be produced. If there is no type expression declared, Catscript will automatically assign a type to the variable based on the value assigned.

```
var x : int = 10
var y = false
```

```
var z = null
```

Function Definition Statement

Function declarations require the function keyword, a function name, a parameter list in parentheses, an optional return type, and the code that the function will run. The grammar structure is shown below.

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                      [ ':' + type_expression ], '{', { function_body_statement
}, '}'
```

If the function is not given a return type, it will be assigned the void type and no return value will be expected. Functions have the option to use return statements to either return a value or to end the function. The examples below showcase different forms of functions, displaying a version with a set return type and a function with the void return type.

```
function final() : int { return 5 }
function foo(x, y, z) { print(x) }
```

Print Statement

The print statement is a simple call that requires the print keyword and an expression within parenthesis. It does not require any form of type expression, only a valid expression to print out.

```
print_statement = 'print', '(', expression, ')'
```

The print statement is able to accept any object value. This can range from strings in the first example below to variable values in the second example. Print statements will get the value of its inputted expression before printing. With the second example, true would be printed out instead of x.

```
print("This is a test")

var x = true
print(x)
```

Return Statement

A core component in functions is the return statement. Return statements require the return keyword and then an expression that matches the function's return type. If the return is either unlisted or declared as void, an expression is not required after the return keyword and instead will end the function and return to where it was called. The two examples below demonstrate return statements that have a return type or return void.

```

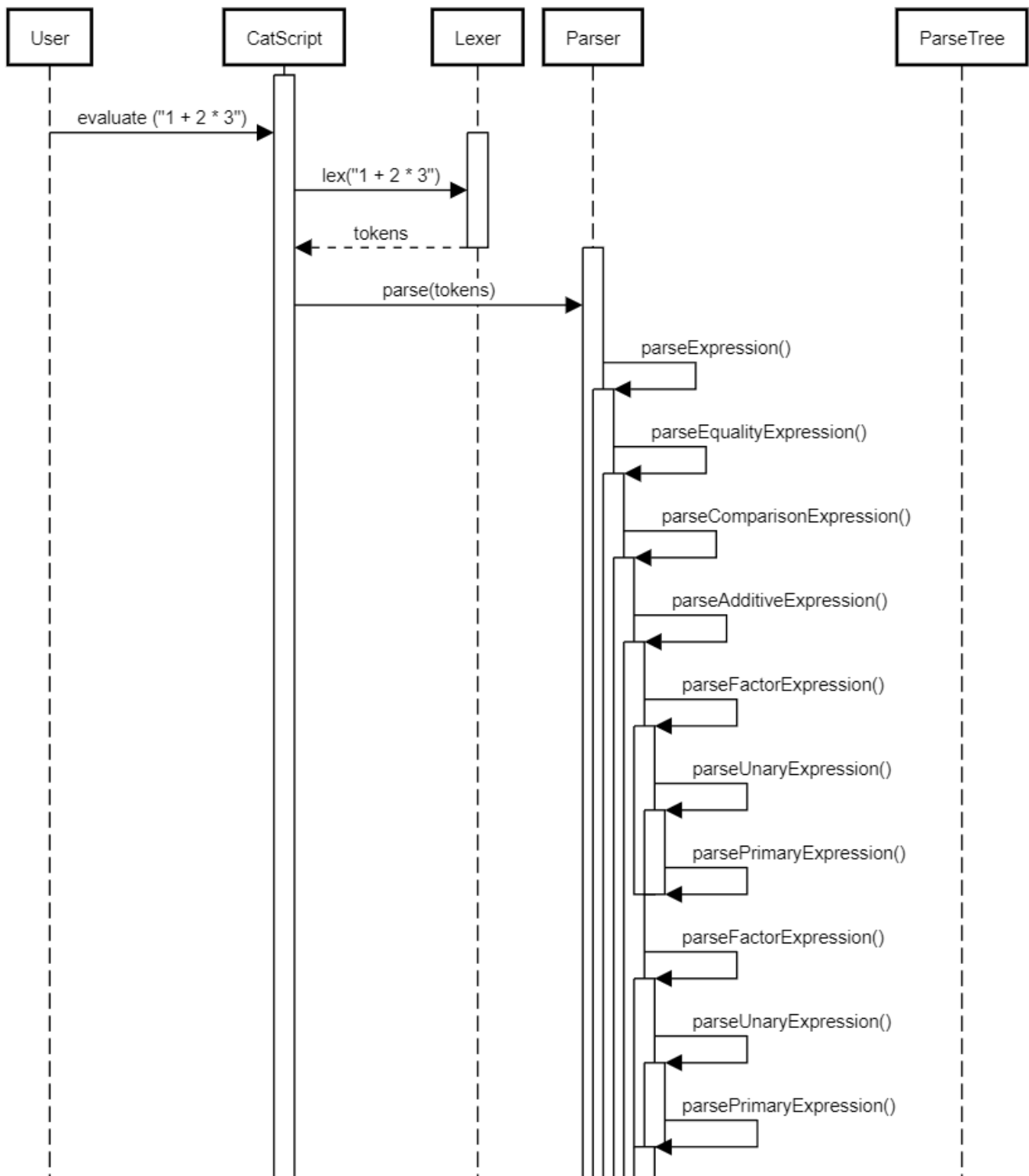
function foo() : object { return 1 }

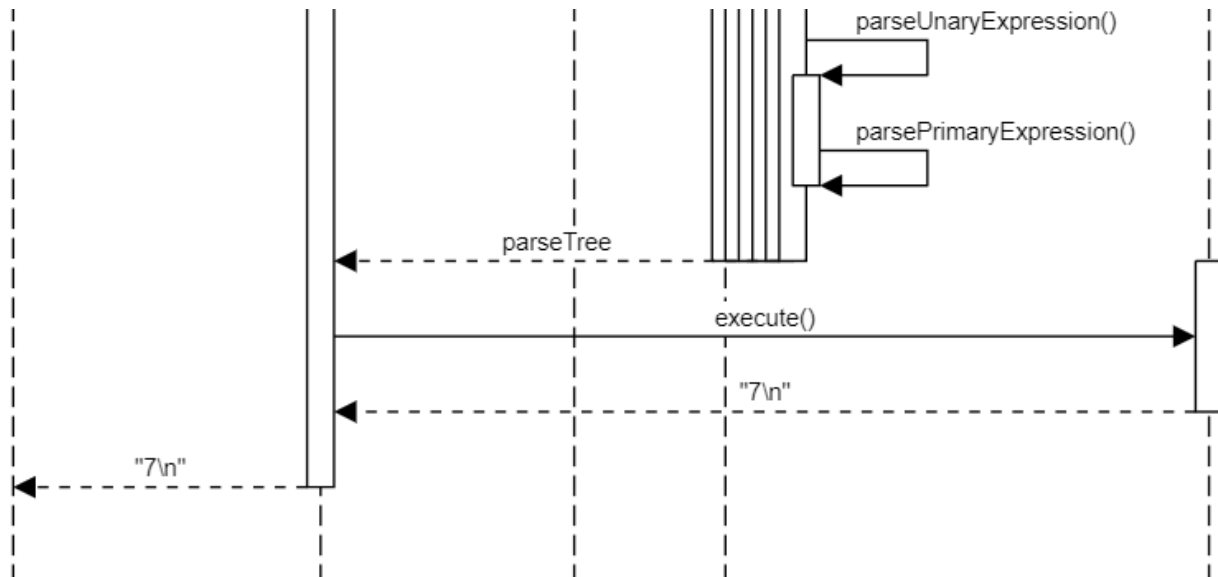
function foo2() {
  print(4)
  return
}

```

Section 5: UML

Catscript Addition and Multiplication Sequence Diagram





Section 6: Design trade-offs

When starting a project, it is important to determine your primary drivers, which will help you make design decisions. We had a few main drivers for this compilers project: implementation complexity, ease of understanding, and time to develop. These drivers are intertwined with each other. We needed to be able to complete this project in a very short timeline (a single semester), so the design had to be quick to understand and quick to develop. Decisions that result in simplistic designs help achieve those goals while complex designs do not. When it came to considering how to generate our Abstract Syntax Tree, considered two different models, Recursive Descent and Parser Generators, and applied our drivers. Recursive Descent parsing is based on the idea that each production in our grammar has a method to parse it. Each method would call the method with the next highest precedence, creating the "recursion," while matching and consuming tokens when appropriate until the entire string/program was parsed. This method was incredibly easy for us to understand, meeting one of our core drivers. It was also very simple and seemed fairly quick to develop, meeting our other two drivers. Parser Generators work on a different principle. They take in a grammar and generate source code for the parser itself. This is great as the main work is generating a grammar, which we need to do for either parsing method. This would be faster than writing an entire Recursive Descent parser as well. However, the major drawback with this method is that we gain little to no understanding of how to parse tokens, as all of that is handled by the Parser Generator. After considering our two options, we decided to go with implementing the Recursive Descent algorithm over using a Parser Generator. The reason for this project is to learn how to build each of the steps of a compiler. The fact that the Parser Generator approach does not lend itself to us learning and implementing a parser itself makes it much less attractive than implementing a Recursive Descent algorithm ourselves.

Section 7: Software development life cycle model

We performed Test Driven Development (TDD) for this project. Before developing the software, we started with an infrastructure with near-zero implementation of any of the logic. This mainly consisted of stubbed-in methods that threw `NotImplemented` exceptions. We then had a large variety of unit tests that were written to test very specific functionality by referencing each of our methods. This allowed us to define what our software was supposed to do instead of how it was meant to accomplish it. When developing the software, Team Member 1 selected a unit test and programmed the related methods and logic in the project until the test passed. He was then able to run all the unit tests, ensuring that none of the previous passing tests failed due to his changes. He then was able to repeat this for every test until all were passing. With these tests, it

was quite easy for Team Member 1 to ensure that any refactoring didn't change the overall output of the program. This was very helpful when implementing the memoization, as that refactor was implemented after all of the unit tests passed. Team Member 1 was able to refactor the related methods without fear of unintentionally breaking anything, as once he was done, he ran all the unit tests to ensure no functionality was broken.