Montana State University

Gianforte School of Computing

CSCI 468 Compilers

Rory Donley-Lovato

Braedon Hunt

May 2023

Section 1: Program

The source code can be found at this location:

Rordog/csci-468-spring2023-private/capstone/src.zip

Section 2: Teamwork

This project was split into two aspects: writing the program and quality assurance. Team Member 1 was responsible for writing the majority of the code and managing the GitHub repository. The code was broken up by checkpoints of completing the tokenizer, parser, evaluator, and bytecode compiler. These checkpoints consisted of numerous tests to ensure functionality and prevent bugs from existing in the final program. Team Member 2 was responsible for writing the tests to create a test-driven development environment and writing the technical documentation. Communication between the team members took place over discord and in person.

Estimated Contributions

Total Estimated hours: 90

Team Member 1:

Contributions: Code management and primary programmer.

Estimated Work Hours: 75 hours, ~83% contribution in time.

Team Member 2:

Contributions: Documentation and tests.

Estimated Work Hours: 15 hours, ~17% contribution in time.

Section 3: Design Pattern

Memoization is the technique of saving the results of a function call so they can be easily accessed by identical calls. It is most commonly used with computationally expensive calls in order to optimize it. By saving the results of such function call, the program does not need to spend time and computation power to compute the results again, instead it just pulls the results from its storage. This storage is often set up in a HashMap or similar data structure. HashMaps are used due to their get and put methods. HashMaps also allow custom parameter values which is perfect for saving arguments to a function call in addition to the return value. When a memoized function receives a call, it checks to see if that specific call exists in the map. If that call has been saved, the value is pulled from storage and returned without further calculation. If the call does not exist in the storage, it will complete the calculations and then save the result into the table.

In this capstone, memoization was implemented in the getListType call shown below. Memoization was used here as a method to gain experience with this technique. The functions present in this compiler are fairly straightforward, leaving very few opportunities to put this technique to use. I chose to implement this technique in getListType because it seemed like the simplest place to implement it with the least risk of causing tests to fail if done improperly.

Usage of Memoization

```
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;
   }
```

Implementation Location: src.zip\...\parser\CatscriptType.java

Section 4: Technical writing

4.1: Introduction

Catscript is a simple programming language that feels similar to Java and Typescript.

Here is an example of a Catscript program:

```
var x = 5
print(x)
```

4.2: Features

While Catscript does not have all the features of a fulled fledged programming language, it does have a few features that make it functional.

4.2.1: Types

Catscript is a statically typed language and has a few different types that can be used in the language. These types are:

1. Integer This type is used to represent whole numbers. It is represented by the keyword `int`. Here is an example of this

   ```
   var x : int = 5
   ```

2. Boolean The boolean type is used to represent truthy values. It is represented by the keyword `bool`. The boolean literals are `true` and `false`. Here is an example of this

   ```
   var x : bool = true
   ```

3. String The string type is used to represent a string of characters. It is represented by the keyword `string`. Here is an example of this

   ```
   var x : string = "Hello World"
   ```

4. Object This type is used to represent an object. It is represented by the keyword `object` or it can be completely omitted. If a variable statement does not include an explicit type, object is the assumed type. Here is an example of this

```
var x = 1
var y = "Hello World"
var z = true
```

5. Void This type is used to represent a function that does not return a value. It is represented by the keyword `void`. Here is an example of this

```
function printHelloWorld() : void {
    print("Hello World")
}
```

6. Null This type is used to represent a null value. It is represented by the keyword `null`. Here is an example of this

```
var x = null
```

7. List<> This is the only type that is not primitive. It is used to represent a list of values. It is represented by the keyword `list` followed by the type of the list inside closed angle brackets. You can nest lists inside of lists. Catscript also supports list literals, a feature that Java lacks. However, you cannot index the list at arbitrary points and the list itself is immutable. An example of using lists would be

```
var x : list<int> = [1, 2, 3]
var y : list<string> = ["Hello", "World"]
var z : list<list<int>> = [[1, 2], [4, 5]]
```

4.2.2 Assignability

1. Integer

   Integers are assignable to the type int and object.

2. Boolean

   Booleans are assignable to the type bool and object.

3. String

Strings are assignable to the type string and object.

4. Object

   Expcilitly typed objects are only assignable to the type object.

5. Void

   Void is not assignable to any type.

6. Null

   Null is assignable to any type.

7. List<>

   Lists can be assigned to the type object and can be assigned to lists of the same type. They are also covariant, meaning that a list with elements of type P can be assigned to a list with elements of type Q if P can be assigned to Q. An example of this would be

```
var x : list<int> = [1, 2, 3, 4, 5]
var y : list<object> = x
```

4.2.3: Conditionals

1. Equality

   Equality is denoted with ==.

2. Inequality

   Inequality is denoted with !=.

3. Less than

   Less than is denoted with <.

4. Less than or equal to

   Less than or equal to is denoted with <=.

5. Greater than

   Greater than is denoted with >.

6. Greater than or equal to

   Greater than or equal to is denoted with `>=`.

7. Not

   Not is a unary operator and is denoted with `not` followed by a boolean expression or literal.

4.2.4: Control Flow

1. If/Else Catscript supports if/else statements. The if statement is represented by the keyword `if`, followed by a condition in parentheses and a block of code in curly braces. The optional else statement is represented by the keyword `else`, followed by a block of code in curly braces. Here is an example of this

```
if (x == 5) {
    print("x is 5")
} else {
    print("x is not 5")
}
```

2. For Loop

   Catscript supports for loops. The for loop is represented by the keyword `for`, followed by a variable name, the keyword `in`, and a list object. The for loop will iterate over the list and assign each value to the variable. Here is an example of this

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

4.2.5: Functions

1. Function Declaration

   Catscript supports functions. Functions are declared by the keyword `function` followed by a function name, a list of parameters in parentheses, an optional return type, and a block of code in curly braces. Here is an example of this

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

2. Function Call

   Catscript supports function calls. Function calls are represented by a function name followed by a list of arguments in parentheses. Here is an example of this

```
multiply(5, 5)
```

3. Function Parameters

   Catscript supports function parameters. Function parameters are represented by a variable name optionally followed by a type. Here is an example of this

```
function divide(x, y : int) {
   return x / y
}
```

4. Function Return

   Catscript supports function return. Function return is represented by the keyword `return` followed by an expression. Here is an example of this

```
function add(x : int, y : int) : int {
   return x + y
}
```

   If a function does not return a value, the return type is `void` and the return statement is optional. Here is an example of this

```
function printHelloWorld() : void {
   print("Hello World")
}
```

   or

```
function invokeFoo() {
   foo()
   return
}
```

4.2.6: Scope

1. Global Scope

   Variables that are declared outside of functions are global in scope. Here is an example of this

```
var x = 5
function foo() {
   print(x)
}
```

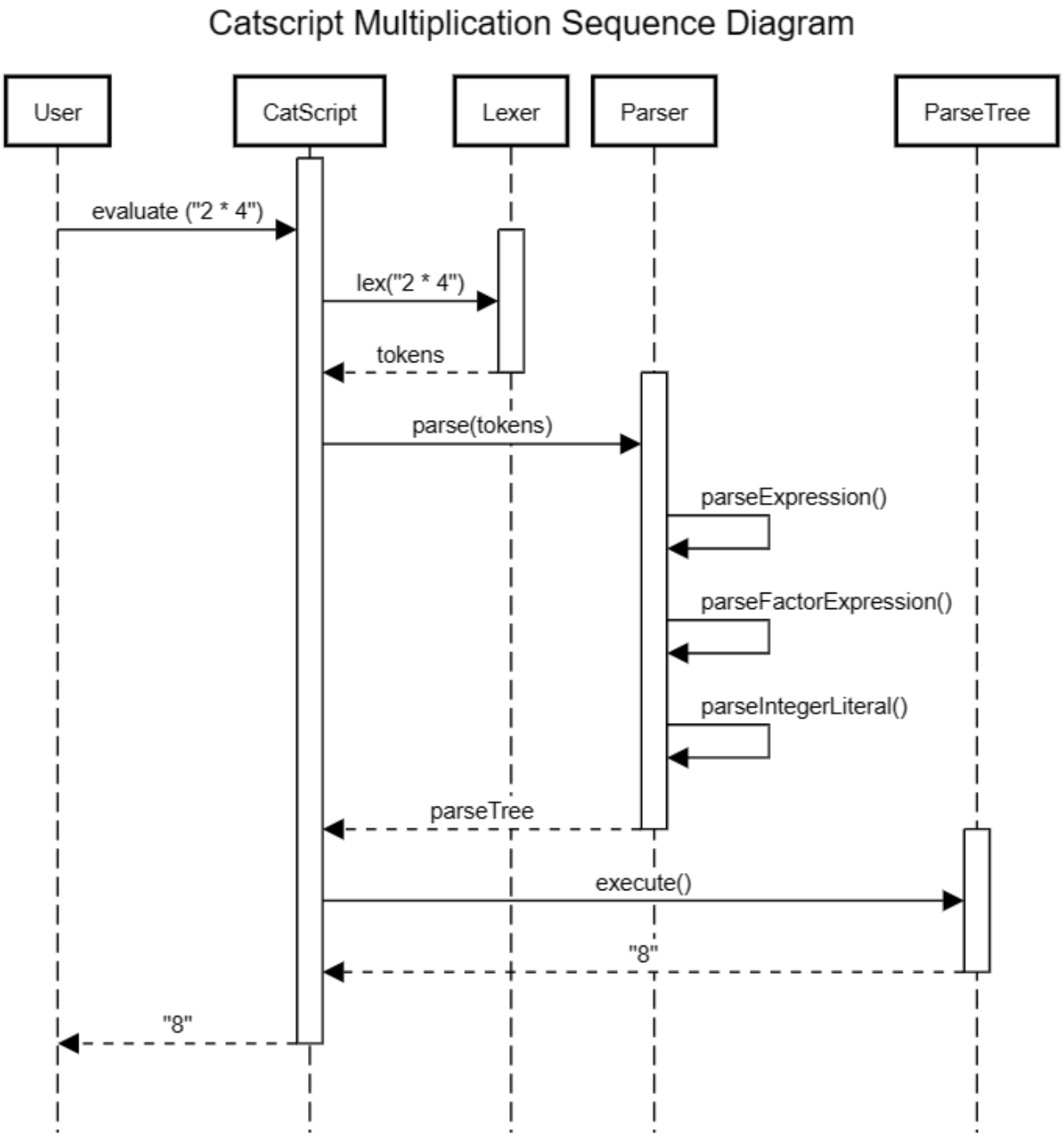
2. Local Scope

   Variables that are declared inside of functions are locally scoped to that function. Here is an example of this

```
function foo() {
   var x = 5
   print(x)
}
```

   The variable x is not accessible outside of the function foo.

Section 5: UML Diagram



Sequence Diagram of Factor Expression Evaluation

Section 6: Design Trade-offs

Recursive Descent vs Parser Generators

Recursive Descent

Recursive Descent is a method of creating a parser where the developer has to create every piece. By making programmers write out the entire parser, the developers have a better understanding of how it works, and it will likely be fairly simple and easy to read. The technique works by having a method for every piece of grammar possible. As it reads the code, the parser will call the corresponding methods that match the incoming grammar. This form of parser does take more time to create than having it generated, but the benefit of having developers who understand it outweighs the cost of time.

Parser Generator

Parser Generators consist of two parts: regular expressions and grammar in EBNF (Extended Braccus Naur Form). The lexer is created by generating code that supports the grammar. The parser is creating in the same way as the lexer and instead an AST (Abstract Syntax Tree). The generated parser is a form of a recursive descent parser. The problem is that it is much more difficult to read because it originates from an abstract form. Due to this, more time is invested in learning to read the generated parser than if it was written by hand. Because of these complications, Parser Generators are not commonly used and are instead replaced by Recursive Descent parsers written by hand.

For the purpose of creating this project, Recursive Descent was used to provide experience with most common form of parser and give a better understanding of how parsers work by writing it ourselves.

Section 7: Software Development Life Cycle Model

Test-Driven Development is a style of programming that bases success and completion off of completing numerous tests. It simplifies how the developers write code by giving them smaller objectives to complete and not allowing them the time to overcomplicate code. It is also tied into quality assurance due to the tests that are designed to ensure functionality of the program.

I liked using this development method. It allowed me to keep the code simple by only creating what was necessary for the tests to pass. The tests allow for easy checkpoints that can separate the program into its individual steps. This causes straightforward development by giving the programmers an order in which to write the code. I can create components in the order needed and not get sidetracked by trying to make everything work at the same time. Test-Driven Development has made this project so much easier because I'm able to debug a piece of the program and ensure that it works. If I wrote this compiler without tests, it would've taken so much longer because I would not have a way to figure out what parts work and what parts are still broken.