

Final Capstone Portfolio

CSCI 468 – Compilers – Spring 2023

Travis Brase

Aurora Duskin

Section 1: Program

Please see included source.zip file for source code

Section 2: Teamwork

This project was split into two main sections with each team member completing their own compiler based on an initial code base provided by the instructor. For the majority of the semester each member worked through a series of checkpoints that evaluated their progress and understanding of the material. Near the end of the semester the team members exchanged their compilers and wrote additional tests and documentation based on the compiler provided by the other team member. Both team members were essential to the completion of this capstone project. Provided below is an estimated breakdown of time spent on the project:

Total Estimated Hours: 60

Code Development 83.33%: (Member 1) 50 Hours split evenly across the 4 sections

Technical Writing 6.67%: (Member 2) 4 Hours

Testing/Debugging 3.33%: (Member 2) 2 Hours

Portfolio Completion 6.67%: (Member 1) 4 Hours

Section 3: Design pattern

We used the memoization pattern a prime example of this starts on line 35 of the CatscriptType.java file of my project. Memoization is basically a form of caching, that allows us to use less system resources when repeating tasks by storing the data and then simply recalling it instead of re-computing again and again. Our implementation is only effective for a single threaded application, however, since that is all that we are working with it is sufficient for this use case.

```
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 new ListType(type);
}

```

Section 4: Technical writing

Expressions

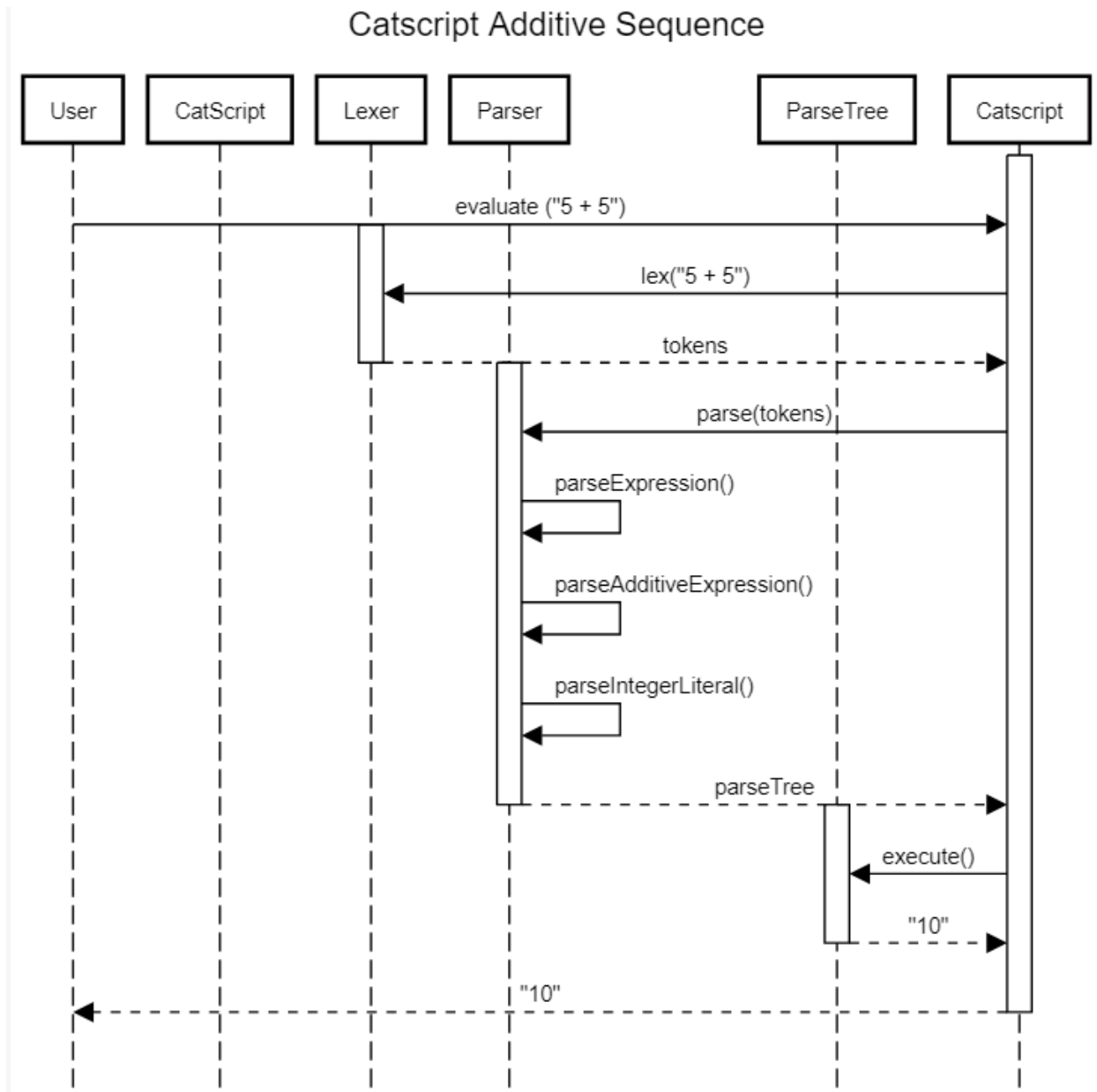
Syntax	Arguments	Return Type	Usage
$x > y$	Int, double, float	boolean	Evaluates TRUE if x is strictly greater than y, else returns FALSE
$x < y$	Int double, float	boolean	Evaluates TRUE if x is strictly less than y, else returns FALSE
$x \geq y$	Int double, float	boolean	Evaluates TRUE if x is greater than or equal to y, else returns FALSE
$x \leq y$	Int double, float	boolean	Evaluates TRUE if x is less than or equal to y, else returns FALSE
$x == y$	Int double, float	boolean	Evaluates to TRUE if x equals y, else returns FALSE
$x != y$	Int double, float	boolean	Evaluates to True if x does not equal y, else returns TRUE
$x * y$	Int double, float	Type of inputs	Multiplies the value of x by the value of y and returns the result
x / y	Int, double, float	Type of inputs	Divides the value of x by the value of y and returns the result
$x - y$	Int, float, double	Type of inputs	Subtracts the value of y from x and return the result
$x + y$	Int, float, double	Type of input	Adds the value of y to x and returns the result
$x + y$	String, int, double, float	String	concatenates the values into a single string literal expression and returns the resulting string

Statements

Syntax	Usage
--------	-------

<code>if(boolean){expression}else{expression}</code>	If the boolean evaluates to TRUE then execute the expression inside the first set of {} otherwise executes expression inside the second set of {}
<code>for(x in array[]){expression}</code>	Executes the expression within the {} the number of times equal to the size of the array. During each loop the matching element of the array may be accessed by an expression using x as the label to access.
<code>print(x)</code>	Prints the value of x to the console
<code>function foo(x){expression}</code>	Establishes foo as a callable function that accepts argument(s) x and executes the expression within the {}

Section 5: UML.



Section 6: Design trade-offs

The biggest trade-off of this class was that we wrote our parser by hand as opposed to a parser generator. We did this because it gives us a more in-depth view of the parser instead of having the generator use it for us. If this compiler were to be written in a real-world environment instead of an academic one it would have made more sense to use

the generator since that would be a more efficient use of actual development time, instead of writing it all by hand.

Section 7: Software development life cycle model

We used Test Driven Development (TDD) for this project. Basically, we were given a large suite of tests that we then focused our efforts around completing those tests. From an academic perspective this makes a lot of sense and is convenient since it provides an easy-to-follow roadmap for the semester and the expectations for the class. However, one potential drawback to this approach is that it makes it possible to target the tests and therefore not complete the spirit of the assignment, as the tests may pass, but the program would be flawed for real-world use. As part of the teamwork aspect of this project we wrote tests for each other. The additional tests that I was provided are as follows:

```
@Test
void testFunctionCallsInForLoop() {
    assertEquals("1\n2\n3\n", executeProgram("function foo(x){print(x)}"+
        for(x in [1, 2, 3]) { foo(x) }"));
}

@Test
void testAllOperators() {
    assertEquals(5, evaluateExpression("2 * 3 / 2 + 4 - 2"));
}

@Test
void testMixAndMatch() {
    assertEquals("2a", evaluateExpression("1 + 1 + \"a\""));
}
```