

**Capstone:**  
**Catscript Compiler**



Zachary Jewett  
Montana State University  
CSCI 468 – Carson Gross  
Spring 2023

## Table of Contents

<a href="#"><u>Section 1: Program</u></a> .....	2
<a href="#"><u>Section 2: Teamwork</u></a> .....	2
<a href="#"><u>Section 3: Design Pattern</u></a> .....	3
<a href="#"><u>Section 4: Technical Writing</u></a> .....	4
<a href="#"><u>Section 5: UML</u></a> .....	6
<a href="#"><u>Section 6: Design Trade-Offs</u></a> .....	7
<a href="#"><u>Section 7: Software Development Life Cycle</u></a> .....	8

## Section 1: Program

The source code, *source.zip*, for the Catscript compiler is located in the directory. The compiler was developed in Java, utilizing recursive descent with the Memoization design pattern, and followed a Test-Driven Development software life cycle. Maven was used for dependency management structure. The foundational development environment is Mr. Carson Gross and Montana State University.

```
src > capstone > portfolio > source.zip
```

## Section 2: Teamwork

This capstone project was completed individually, as I failed to pair with a partner. Recorded lectures, instructor published slides, instructor help sessions, and peer Discord communications proved instrumental in progress of the Catscript compiler. Testing and quality assurance was intended to be based on partner feedback and documentation, however that is, as of currently, an unmet requirement on my end.

## Section 3: Design pattern

### Memoization Pattern

In its most succinct definition, memoization is a technique to improve program performance. It does so by caching previous calculated results to be used again in recursive algorithms. By sacrificing some memory, we can avoid redundant calculations and reduce time complexity. While not itself a specific design pattern, Memoization is a common technique in dynamic programming for optimization. It is easy to implement into specified design patterns such as the Decorator or Singleton patterns.

In the CatScript compiler, we utilize 1D-Top-Down memoization. We break down the large problem of compiling a language into smaller, deterministic function calls relying on recursive descent, which will be further discussed in section 6. Utilizing the code below, found on line 37 in CatscriptType.java, the memoization technique calculates a function once and then stores the resulting CatscriptType. Anytime a memoization function is called, the “cache” is searched, and if found, returns the stored result without expensive recalculation. Otherwise, the function is copied and stored for future use. Specifically, this memoization code uses a HashMap with a CatscriptType serving as both the key and the value. The purpose is to return a ListType object that matches the input.

```

36      //Cache implementation for Memoization Optimization
      2 usages
37      static Map<CatscriptType, CatscriptType> cache = new HashMap<>();
      6 usages  ↳ AlngCameAWizard +1 *
38      @ public static CatscriptType getListType(CatscriptType type) {
39          CatscriptType match = cache.get(type);
40          if(match != null){
41              return match;
42          }
43          else{
44              ListType listType = new ListType(type);
45              cache.put(type, listType);
46              return listType;
47          }
48      }

```

```
src > main > java > edu.montana.csci.csci468 > parser > statements > CatscriptType.java
```

## Section 4: Technical Writing

\*Disclaimer: *“Include the documentation generated by your partner for the Catscript programming language”* I never paired with someone. In light of this missed requirement, this section was completed solely by yours truly.

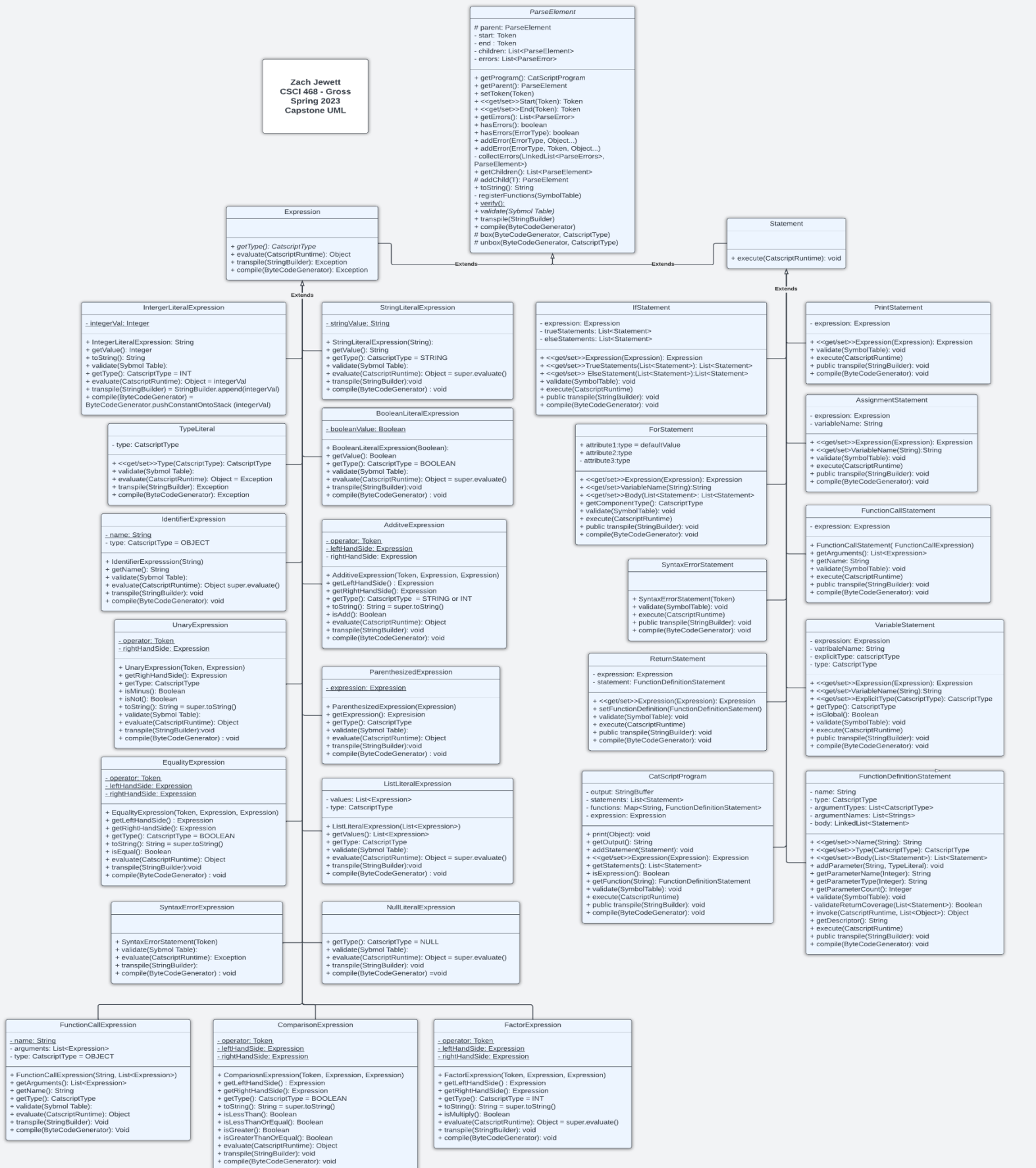
**Tokenization**

**Expression Parsing**

**Statement Parsing**

## Section 5: UML

Zach Jewett  
CSCI 468 - Gross  
Spring 2023  
Capstone UML



## Section 6: Design Trade-Offs

### Recursive Descent vs. Parser Generator

A very adamant decision on the part of the instructor was to utilize recursive descent over a parser generator. A decision founded on simplicity and practicality. It proved to be the correct choice. Recursive descent is a top-down parsing technique that goes hand in hand with the memoization design pattern mentioned in section 3. Parsing begins with taking a sequence of tokens from a lexical analyzer as an input and building a parse tree from the derivations. Recursive descent breaks down the complex problem, like compiling an EBNF language grammar, into a series of smaller subroutines resulting in this tree-like syntax structure. Each simplified subroutine handles production for non-terminal symbols. In turn, each non-terminal symbol is composed of terminal symbols that make up the smallest units of input that can be recognized. Recursively iterating through the terminal symbols to construct a non-terminal, and then repeating that process until an input consists of only non-terminals leads to a successful parse. While a little jargon heavy, implementation is intuitive and explicit.

It is in this simplicity that recursive descent overcomes a parser generator. Parser generators automatically produce code to implement a parsing algorithm for the target language. Only requiring two inputs via regular expression (REGEX) and a Backus-Naur Form (BNF/EBNF) language grammar, parser generators are convenient and less susceptible to human error. However, each parser generator typically requires following their own specialized language syntax, making quick adoption difficult. More significantly, they are extremely tedious to debug and maintain as internal states may limit visibility and readability is hindered. In contrast, recursive descent code is much more readable and works with built-in IDE debuggers. Furthermore, the looping control flow for descent parsing is easy to follow, maintain, and extend. We do hit a snag if the grammar contains any left-recursion.

+ Recursive Descent -		+ Parser Generator -	
Simplicity	Limited Grammar Complexity	Efficiency	Steep Learning Curve
Debugging	Code Duplication	Automation	Limited Flexibility
Performance	Error Handling	Consistency	Debugging

## Section 7: Software Development Life Cycle Model

### Test-Driven Development

The Catscript capstone project was built following Test-Driven Development (TDD) software development life cycle design. TDD uses automated tests to provide targets and validation for developers. These test cases are written before actual code production begins and bestow a sense of direction to programmers. Further, multiple tests groups provide solid checkpoints for developers and allow them to evaluate their progress. Successful completion of all the tests should mark completion of the program. Ultimately, TDD limits bugs, improves code quality, and allows faster development. The Catscript compiler tests were developed by Mr. Carson Gross.

From a personal perspective, Test Driven Development is phenomenal. Having all the development goals laid bare is incredibly helpful for someone like me who will often go off on a programming tangent of some small idea. It is the most satisfying feeling to see all those green checkmarks, but also the most depressing when something fails, over and over. Indeed, as I failed to reach certain milestones, and then skipped ahead to meet the next one, I would leave “broken windows” in my wake. This caused cascading failures as my subpar foundations would not properly fit with the next steps. This would be accompanied by a waterfall of refactoring and frustration. So, Test-Driven Development is perfect for the punctual.