

## Section 1: Program

The source code for the compiler is included in the attached zip file.

## Section 2: Teamwork

Compilers are complex systems, and it is necessary to understand the entire process to be able to build a compiler. For that reason, the compiler was designed and programmed without partner help. If this were to have been a group project, many groups would have one member doing most of the work and the other member would have passed the class with no clue of how to design a working compiler.

The teamwork in this project consisted of test-driven-development and documentation. For each two-member team, each member wrote tests and documentation for the other member's compiler. This allowed team members to check their partners work without the risk of one person coasting.

## Section 3: Design pattern

The Memoization pattern was used to optimize the `getListType()` function in the `CatScriptType` class. Initially, this function created a new (immutable) object each time it was invoked, resulting in overhead each time it was called. This wastes both CPU time and memory and places unnecessary stress on the JVM's Garbage Collector.

```
// The below Map serves as a cache to implement memoization.
// Once a final object is created once, it never needs to be recreated.
private static final Map<CatscriptType, ListType> catscriptTypeCache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    // If the memoized value does not exist, create it
    if(!catscriptTypeCache.containsKey(type)){
        catscriptTypeCache.put(type, new ListType(type));
    }
    // Return the memoized type
    return catscriptTypeCache.get(type);
}
```

## Section 4: Technical writing. Include the technical document that accompanied your capstone project.

The included documentation file 'catscript.md' explains the grammar and functionality of the CatScript programming language.

## Section 5: UML.

CatScript uses a parser to generate a parse tree. The parse tree is formed from ParseElements that follow a hierarchy. This parse tree is used for evaluation, bytecode generation, and (not implemented) transpilation to JavaScript.

UML Diagram for ParseElement.java:

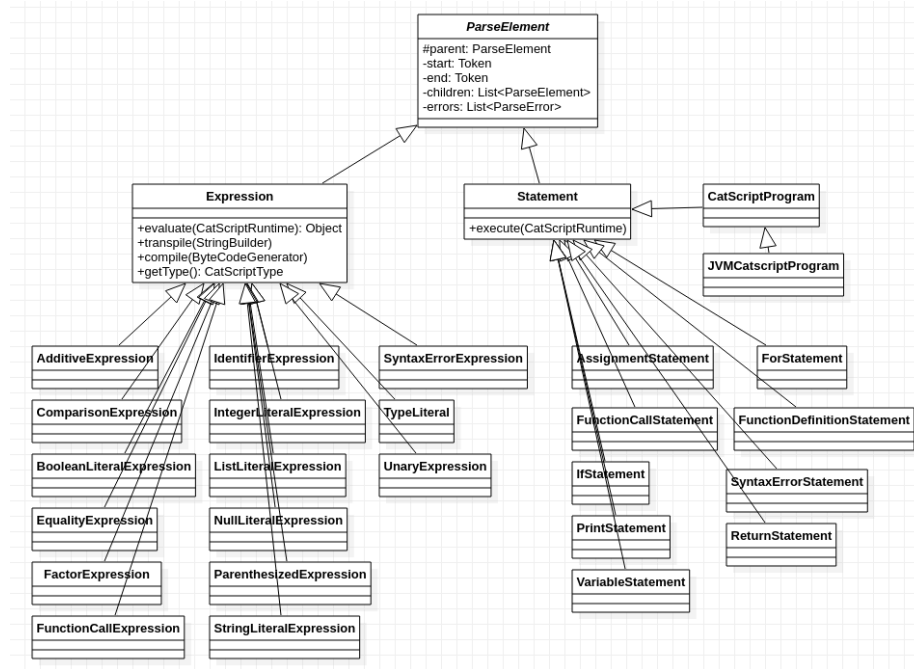


Figure 1: ParseElement UML

## Section 6: Design trade-offs

In general, compilers can be created in one of two ways, Parser Generators and Recursive Descent. Parser generators are often used due to the ease of creating a working compiler, but they are sub-optimal for the end user. The code they generate produces confusing error messages for the end user, and they are difficult to debug.

We chose a recursive descent compiler to satisfy the above-mentioned learning and debugging criteria. The recursive descent pattern is similar to how the other parts of the compiler (evaluator and bytecode generator) work and gave me a better understanding of how compilers actually work.

## Section 7: Software development life cycle model

In this class, we used test-driven development (TDD). Test-driven development starts with a collection of tests that the software must pass and a “framework” application with few features implemented.

Each stage of the compiler had its own set of tests. The four sections of the compiler (Tokenizer, Parser, Evaluator, and Bytecode generator) each had their own series of tests.

Test-driven development helped greatly with developing all parts of the compiler. Since the program specification was specified by programmatically executed tests, it was easy to see which parts of the program were functioning properly, and which parts still needed work. Additionally, TDD provided an easy way (regression testing) to check if new code introduced bugs by re-running previous tests.