

## Section 1: Program

---

A zip file of the source code is included in this directory in [source.zip](#).

## Section 2: Teamwork

---

For the capstone project, our team separately used test driven development to develop separate recursive descent parsers. Both partners also wrote the technical documentation for the other partner's project. We each wrote tests to assist each other in writing a fully complete parser. My partner's tests are provided in [src/test/java/edu/montana/csci/csci468/demo/PartnerTests.java](#).

## Section 3: Design pattern

---

The Memoization Pattern is used to memoize calls to [CatScriptType#getListType\(\)](#). If we are creating a list we don't want to store the type of each element with the element itself more than we have to, so instead we use our design pattern. A hash map is used as a cache to store different variants of list types. This frees up memory and sufficiently implements the Memoization Pattern

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = cache.get(type);
    if(listType == null){
        listType = new ListType(type);
        cache.put(type,listType);
    }
    return listType;
}
```

## Section 6: Design trade-offs

---

We used a handwritten recursive descent instead of a parser generator to develop our catscript parser. Parser generators are the typical method for creating a parser in a compilers course. All they require is an input grammar, and most of the course would have been dedicated to developing that. It's simple to write a specification if the input format is close to normal. The end result is typically easily maintainable and understood. These parsers are also faster than hand written ones, but not in the case of handwritten recursive descent parsers. However, parser generators can sometimes reject grammars so research into the specific generator must be done before hand.

A handwritten recursive descent parser comes with many advantages. The programmer gains a deeper understanding of all components, the parser is faster because it begins at the start symbol of the program with no back tracking, the output creates a parse tree, a incredibly useful and fast data structure. However, all the code had to be handwritten, and the complexity of the project was high. A tokenizer, parser, evaluator, and bytecode generator all had to be implemented which took significant effort. This method is also less space efficient than other methods due to the large amounts of function calls and recursion.

## Section 7: Software development life cycle model

---

We are using Test Driven Development (TDD) for this project. TDD is a software development approach where test cases are created to specify and validate every small functionality of the program. Tests are tested first, and if they fail new code is written to fulfill them. At the beginning of the project having so many tests to pass was daunting. Even finding where to start was troublesome. However, after the first checkpoint this is our new favorite life cycle model. What was unclear and insurmountable and the beginning became a task list of little check boxes. Easily allowing us to isolate a specific piece of code to implement correctly before tackling the rest of the project. Had the development model been more traditional to university courses having to write the thousands of lines of code required would have been impossible.