

Section 1: Program

[Link to the zip file containing my capstone project.](#)

Section 2: Teamwork

With my partner's Catscript documentation, there were a lot of important pieces of the documentation already in place. However, there were some big picture / general structure pieces that were missing or needed to be expanded upon. These were the biggest changes I suggested for my partner. In addition to those ideas, I also thought the wording of some sections, especially those about functions needed more details to describe all situations. The main overlooked areas were with explicit typesetting, which my partner fixed after we talked about it. Lastly, I thought that some examples could use comments within the code to explain what was going on to just give additional context with what was going on. Once all of these changes were implemented, I think my partner's documentation ended up being very well fleshed out and fully realized. Their naturalistic tone throughout could have been maybe a little more formal, though that's not a big deal. I still think some explanations could be a little more clear with functions, but they are much better and still work well enough in the final result.

As for my partner's tests, they focused on ensuring type setting is generalized, list types are assigned recursively, and function return coverage is assigned recursively. These tests were well-designed already, but I did suggest a couple changes to improve them. With both of the recursive cases, I recommended adding an extra recursion layer to both tests. To pass the generalization, I checked the type of each object the list stores: if all the items were the same type, that type would be used, otherwise the object type would be used. Next, I had to change my list type setting into a recursive method. I was already using a method called `getCatscriptType` to find the type of an object, but it wasn't recursive. I changed the method to be recursive when a list type's explicit type is given. Last, I edited my function coverage method to recursively find if each statement has valid return coverage, allowing multiple nested if statements.

In terms of the tests I wrote for my partner, I focused on overlooked functionality and edge cases that weren't covered in the original documentation. For overlooked functionality: I created a test for string addition since not all types of string addition were tested for in the original parsing testing. When strings are added to another type of value, the other value is converted into a string and the two pieces are concatenated together. The original tests covered the addition of strings with ints and null, but lists and booleans are not. I added these two other string addition cases as one of my tests. Next, I made a test to check for edge cases with comma use in lists, function definitions, and function calls. In all three of these, the original test suite does not ensure that there must be commas between every item and there cannot be a comma after the last item. I created a test that made sure this was the case in all three situations. Last, I tested for the edge case of a function with a void return being stored as a variable. Void should not be able to be stored, so my test checks for a type mismatch error when this situation occurs.

Section 3: Design pattern

One design pattern I used in my capstone project can be found in [CatscriptType.java](#) in the `/src/main/java/edu/montana/csci/csci468/parser` directory. The code for this pattern can also be seen copied below:

```
// Memoization Pattern: when something is expensive to compute based on some arguments
//    So use arguments used to compute expensive thing, and then keep a record
static Map<CatscriptType, CatscriptType> MEMOIZATION_CACHE = new HashMap<CatscriptType, CatscriptType>
public static CatscriptType getListType(CatscriptType type) {

    CatscriptType existingType = MEMOIZATION_CACHE.get(type);
    if (existingType != null) {
        return existingType;
    }
    else {
        MEMOIZATION_CACHE.put(type, new ListType(type));
        return MEMOIZATION_CACHE.get(type);
    }
}
```

I used the Memoization pattern in this specific case for performance reasons. It works by storing the results of an expensive function call in a cache so it can be accessed if the same call is repeated. This doesn't improve performance for the initial call, but for all calls that perform the same action, the performance will be much faster since they can simply access the cache. In this specific case, memoization was used for the `getListType` method since this method could become expensive if used repeatedly.

To implement this pattern, I started by creating a static map that could be used as the cache for this method. The cache is static for two reasons: the method accessing the cache is also static, and so the cache will be accessible for all `CatscriptType` objects. Inside the method itself, I altered the method to first check if a value was cached for the type being passed in. If there was already an item in the cache, that can just be used instead of running the expensive method to get a list type. If there was no item in the cache, a new list type is created for it and put into the cache before being returned.

Section 5: UML

My UML diagram is a sequence diagram modeling the parsing of the expression "printFrom(1 + 2, num)". I chose a sequence diagram because I wanted to see how functions accessed and interacted with each other as they parsed an expression. In this case, I chose a function call containing both an additive expression and an identifier expression. This unique combination causes a recursive call for the two expressions in the printFrom function call, returning both regular and literal expressions.

Section 6: Design trade-offs

In terms of recursive descent versus parser generators for compilers, there are clear advantages and disadvantages for both. Despite these differences, both work very similarly functionality wise: starting by generating tokens, parsing the tokens, creating a parse tree, evaluating the parse tree, then compiling into byte code. For recursive descent, it is much closer to the process of compilation in terms of how it is written. Each part of the compiler must be written in code by hand. Also, as its namesake suggests, it operates recursively to perform the different stages of compilation, which closely reflects the inherently recursive nature of coding grammars. This teaches more about the process of compilation through the writing of the compiler: breaking each step down into understandable chunks. The code is also easy to debug and understand: being significantly shorter, having clear variable and function names, and having comments. Due to these benefits, recursive descent is the method used in industry. However, this methodology is a double-edged sword, as having to write everything by hand means working at a low level in many cases. Recursive descent also requires more infrastructure upfront to make sure each piece is working properly. Overall, recursive descent is a more difficult approach than parser generators, but it creates a clearer understanding of compilers with a better end result.

As for parser generators, they are more frequently used in education as opposed to industry. They rely on lexical grammar and language grammar to generate a parser. This expedites the process of writing a compiler since it needs less handwritten code and doesn't require up-front infrastructure like recursive descent does. It's a much simpler process that is shaped by the definition of file configurations and logic, similar to what can be seen with BNFs and EBNFs. On the other hand, there is a reason why parser generators aren't used in industry. First, parser generators are inherently more declarative because of their creation process, which makes their underlying systems more abstract and unclear. Second, there are many parts of parser generators that are not directly related to compilers, but instead tie into learning a specific parser generator or BNF concepts. This causes parser generators to stray further from their inherent purpose and functionality. Third, code made by parser generators is near impossible to debug or change. Parser generator code is much longer, contains arbitrarily named variables and methods, and lacks any guidance in terms of comments. While parser generators are used frequently in academia, they are poor choices for creating a compiler because of how much they abstract the compilation process and the obfuscated results they produce.

Section 7: Software development life cycle model

We used Test Driven Development (TDD) for this project, which is heavily based on passing tests. This model has both been very useful and somewhat half-baked for the development of this project. In terms of positives, TDD makes debugging code extremely easy since it focuses on the results of different functions. It makes it immediately clear if something is not working properly, and shows with the result in which ways it is off. This can easily act as a springboard for debugging a project, since its clear what part of the program is failing. In addition, it provides guidance for what the result of method should be in the first place. This can act as a nice starting point for development, drawing a clear through line of what a certain method should accomplish.

On the other hand, TDD can also be finicky. Without proper tests, TDD can easily miss edge cases that may be important or even necessary for other parts of the project. This occurred multiple times throughout the development of this capstone project for me, since object type and location setting was not always checked and accounted for in the current testing suite. This would sometimes require major reworks of parts of the project to allow coherence of standards between portions. In addition, this segmentation of TDD generally makes the testing of software interactions harder. While this would still be an issue with other development systems, it remains a problem in TDD.

Overall, I'm glad that TDD was used for the development of this capstone. While it certainly has a couple of small drawbacks, the problems it has are shared by most other development strategies. In addition, the ease of debugging and clear goals that are set up by TDD make progress faster and simpler. Plus, most issues with TDD can be helped or even completely fixed by simply writing more complex tests to check more extreme edge cases.