Example Project Report

Basic Testing Compliance
As per my plan in basic testing, I wrote unit tests for each one of the capabilities. I utilized Rust's built-in handling for unit tests (**https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html**). Unit tests for each component were separated into separate files. These files did not exist when I started on this project; as evidence, here is a link to the commit reflecting the project when it started: **https://github.com/CSUN-COMP587-F18/calculator/tree/4c6d217de74a7c22789a79052583212efa53a842**

Links to unit test files by component follow:
- **Lexer**: Breaks down expressions into a sequence of tokens - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/lexer/test.rs**
- **Parser**: Converts sequences of tokens into abstract syntax trees representing arithmetic expressions - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/parser/test.rs**
- **Interpreter:** Evaluates arithmetic expressions - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/interpreter/test.rs**

For each capability, a link to at least one relevant unit test follows:
- **Lexer** is **Safe**: The lexer never crashes, and delivers informative error messages on invalid input. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/lexer/test.rs#L48**
- **Lexer** is **Correct**: The lexer correctly tokenizes valid input, yielding tokens. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/lexer/test.rs#L58**
- **Parser** is **Safe**: The parser never crashes, and delivers informative error messages on invalid input. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/parser/test.rs#L139**
- **Parser** is **Correct**: The parser correctly parses valid input, yielding an abstract syntax tree. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/parser/test.rs#L62**
- **Interpreter** is **Safe**: The interpreter never crashes, and delivers informative error messages on invalid input (e.g., division by zero). - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/interpreter/test.rs#L43**
- **Interpreter** is **Correct**: The interpreter gives correct answers for valid input. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/interpreter/test.rs#L10**

Advanced V&V Compliance
As per my plan in advanced testing, I wrote fuzzers for each component, and used these fuzzers to try to reveal any crashes in the SUT. Links to the fuzzers are below, on a per-component basis:
- **Lexer**: a fuzzer to automatically generate different character sequences. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/lexer/fuzzer.rs**

- **Parser**: a fuzzer to automatically generate different token sequences. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/parser/fuzzer.rs**
- **Interpreter**: a fuzzer to automatically generate different abstract syntax trees. - **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/interpreter/fuzzer.rs**

Surprises and Challenges
I was already familiar with the codebase and Rust. This helped ensure that there were no major surprises or challenges while I was performing the basic testing. The biggest surprise is that Rust's multiplication will trigger the program to crash if overflow occurs, which is a nice language feature.

The advanced testing was a bit more difficult than I expected, and required an external dependency (in `Cargo.toml`) in order to handle random value generation.

Evidence of Software Improvement
The fuzzer for the interpreter found an entire class of bugs in the interpreter. The interpreter implicitly assumed that none of the values involved in the user expression would overflow. When running the fuzzer, one of the tests caused the interpreter to crash with a multiplication overflow. I fixed this bug by using Rust's checked operations (**https://doc.rust-lang.org/std/primitive.i32.html#method.checked_add**) instead of the usual unchecked operations, allowing me to gracefully handle exceptional values without crashing.

Known Testing Weaknesses
There are not a ton of unit tests written. In many cases, I have only minimal tests checking for error conditions, particularly in the parser. More tests should likely be written. If I had used a code coverage or mutation analysis tool, I would have a better understanding of exactly where more tests are needed, if any at all.

The fuzzers only test with relatively small inputs. It would have been better to test with larger inputs, but no such experimentation was done.

Because the fuzzers only test for crashes, they are fairly limited. I'm not even sure if the parser *can* crash, based on how I've written the code.

Because the fuzzer for the lexer generates random characters, I suspect that the vast majority of the time, the fuzzer generates characters that the lexer immediately rejects as invalid. As such, the lexer fuzzer doesn't deeply test the lexer. In fact, given the class of bugs found in the interpreter, I'm fairly convinced there _is_ a bug in the lexer that the fuzzer is missing. Specifically, look at this line: **https://github.com/CSUN-COMP587-F18/calculator/blob/master/src/lexer/mod.rs#L68**. This multiplication can overflow, and is controlled by user input. However, this can only overflow if a significant number of digit characters are placed together, and the odds of this happening are astronomically low.

Lessons Learned

- I have more familiarity with Rust than when I started.
- Writing a fuzzer that generates completely random test input isn't particularly difficult. However, utilizing that input effectively (i.e., by asking what the input *should* do) is difficult.
- Potentially lots of unit tests are needed to cover even relatively simple code.
- Related to the previous point, code complexity is poorly correlated with the length of the code. For example, the parser is under 200 lines of code, but the entire codebase is a mess of mutually-recursive functions, and it has to track a fair amount of state.