Homework 12

Due Sunday, Nov 11 by 10pm

Turn-In Instructions

Commit your work to your Github project-specific repository. Please be sure that work is committed to a branch called `mostly-working`.

To create and switch to a `mostly-working` branch:

1. Run `git checkout -b mostly-working`, which will create a new branch called `mostly-working`.
2. Make corrections to your assignment's solution.
3. Run `git add` and `git commit` as appropriate to save your changes.
4. To push to `evolene`, run `git push --set-upstream origin mostly-working`. We need to push differently than usual because the `resubmission` branch you just created does not exist on Github. The above command tells `git` to create the branch on Github if it does not already exist.
5. Go to your repository and verify that your new `mostly-working` branch appears in the web interface.

For this checkpoint, your implementation should start looking like what you want it to look like in the end.

Your project implementation should adhere to the following running convention. You should be able to `cd` into the `code` directory and then run your language implementation by typing the command "`dotnet run <args>`". Depending on the design of your implementation, `<args>` should either be a string representing a program or a path to a file containing a program. Running "`dotnet run`" command *without arguments* should make it clear how to call your program with arguments.

This assignment is due on Sunday, Nov 11 by 10pm.

Sanity Check: Students sometimes submit incomplete assignments, accidentally forgetting to run `git add` for all of their files. Fortunately, there is an easy way to make sure that this does not happen to you. Before you are done, `git clone` your repository to a new folder and then try building/running everything. It only takes a couple minutes and can spare you from headaches later on.

Honor Code

This is a partner project lab. You may work with another classmate if you wish, and you may co-develop solutions. Unlike with previous partner labs, you may share source code, and you only need to submit it once. In other words, there is no need to type up and submit your solution twice. As I already know who your partner is, you do not need to submit a `collaborators.txt` file to your repository. However, if your partner arrangement changes, please let me know.

Problems

Q1. (10 points) ..... Execution

Each of the examples described in your last project checkpoint should run and produce the outputs described in your project proposal. In addition, if a user makes reasonable attempts to use your language by referring to the language documentation, those examples should work or mostly work.

At this point, students often realize that their initial vision of their language is not going to work out. Maybe the language is hard to parse, or some detail had not been thought out. That's OK! Change them, if needed to make them work, and if you need to cut a corner or two, that's also OK. Just be sure to make a note that you will need to come back later and make them work for the final submission.

Q2. (10 points) ..... Tests

This submission is required to have at least one test. The required test should be an "end-to-end" test that ensures that for a given program in your language (and user-provided input, if your language needs them) you get a given output. The precise content of these tests is up to you, because it's your language, but you must have one or more. To be clear, the test should check that a parsed and evaluated input produces an expected output.

Your final project will require more tests, at least one for each evaluation rule in your `eval` function. If you want to get a head start, work on those tests now. From personal experience developing languages, I view tests as a time-saver and not a time-waster. It is always frustrating to discover that a newly-added feature breaks other functionality. Making that discovery well after you've added the feature—that's even worse. Having a good test suite will help you find problems early, and it will save you a lot of sweat and tears.

You might also consider testing your parsers, which are pure functions in your language implementation, and therefore "easy" to test. For example, each subcomponent of a parser is itself a parser, and since parser combinators are pure functions, they can be called independently of each other. To test parsers, you will first need to `prepare` your input string, then pass it to one of your parser functions, then check for `Success` or `Failure`.

I should be able to run your one test by running `$ dotnet test` in the directory in which your `sln` file resides. Once you have additional tests, I should be able to run those the same way.