Lab 10

Due Monday, May 13 by 11:59

Handout 27
CSCI 334: Spring 2024

Turn-In Instructions

For this lab, you will continue to work with your existing project repository. Be sure to follow the instructions for committing your file to the appropriate branch.

Turn in your work using the `git` repository assigned to you. The name of the `git` repository will have the form `https://aslan.barowy.net/cs334-s24/cs334-project-<USERNAME1>-<USERNAME2>.git` For example, if your CS username is `abc1` and your partner's is `def2`, the repository would be `https://aslan.barowy.net/cs334-s24/cs334-project-abc1-def2.git`

Pair Programming Assignment

This is a pair programming lab. Like previous partner labs, you may work with a partner. However, for pair programming assignments, you may collaborate to produce a single solution. You do not need to submit a `collaborators.txt` file for this lab.

This assignment is due on Monday, May 13 by 11:59.

Reading

1. **(Required)** Read "The Rise of Worse is Better" from the course packet. Consider this reading to be helpful (and amusing) advice from an expert developer about developing large software projects.
2. **(Required)** Read "Unit Testing in F#" from the course packet.
3. **(As needed)** Read "Implementing Scope" and "Implementing Functions" from the course packet if your language makes use of these features.

Problems

Q1. (10 points) .... **Set the Project Branch**

Your work must be committed to a branch called `mostly-working`.

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

(a) Run `git checkout -b mostly-working`, which will create a new branch called `mostly-working`.

(b) Make your changes, then `git add` and `git commit` as appropriate to save your changes.

(c) To push the new branch to `aslan` for the first time, run `git push -u origin mostly-working`. We need to push differently than usual because the `mostly-working` branch you just created does not exist on the server. Subsequent calls to `git push` can be made as usual.

(d) Go to your repository and verify that your new `mostly-working` branch appears in the web interface.

After you have pushed the `mostly-working` branch to the server, if your partner wants to check out the same branch, they should first `git pull` and then run `git checkout mostly-working`.

Q2. (40 points) .... **Mostly Working Project Prototype**

In this assigment, you will build a "mostly working" version of your language.

Your goal for this assignment is to implement the parts of your language needed to run the examples described in your last project checkpoint. They should produce the outputs described in your project proposal. It's OK if you can't get all of the examples working, but I would like to see you make a strong attempt at each one.

At this point, students often realize that their initial vision of their language is not going to work out. Maybe the proposed language is hard to parse, or some detail had not been thought out. That's OK! You are free to change the language syntax (and, consequently, the examples) to make it easier to parse, and if you need to cut a corner or two in the evaluator (e.g., ignoring certain cases), that's also OK. Just be sure to note which elements need further work for the final submission.

Be sure to see the last question in this handout for the organization of code for this checkpoint. Remember that you must use F# for your implementation.

Q3. (30 points) .... **Minimal Semantics**

This version of your project should explain the semantics of at least two of the constructs in your program. *A language semantics explains how syntax is converted into an AST node (or nodes) and what that AST fragment means.* A good choice at this stage is to describe one of your language's primitives (e.g., data) and one of your language's combining forms (e.g., an operation). For example, you might build the following table.

Syntax	**Abstract Syntax**	**Prec./Assoc.**	**Meaning**
`<n>`	`Number of int`	n/a	n is a primitive. We represent integers using the 32-bit F# integer data type (`Int32`).
`<expr> + <expr>`	`PlusOp of Expr * Expr`	1/left	`PlusOp` evaluates two expressions, $e1$ and $e2$, adding their results, finally yielding an integer. Both e_1 and e_1 must each evaluate to `int`, otherwise the interpreter aborts the computation and alerts the user of the error.

Your semantics does not need to be formal, and it does not need to be in a table, but it should discuss the items shown the table above. For your final project, you will be required to document all of the parts of your language, so if you want to get a head start, you may describe more than two constructs here.

Your semantics should be added to the specification document that appears in your "`code`" folder. You must use LaTeX for your specification.

Q4. (20 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 the test is up to you, because it's your language, but you must have at least one test. 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, you may work on more 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 and relatively "easy" to test. 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` in your test.

I should be able to run your one test by running `$ dotnet test` from your `code` directory. Since tests run in the folder that your `sln` file resides, you will need to reorganize your project. Your solution should be in the `code` folder, your language implementation should be in a subfolder, like `code/lang`, and your tests should be in a folder called `code/tests`. See the reading on tests in the course packet for more details about project organization.