Homework 10

Due Sunday, May 15 by 10pm

Turn-In Instructions

Since you will be committing your work to an existing repository, please commit your work to a new branch called `mostly-working`.

This assignment is due on Sunday, May 15 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) ..... Language Name

If your language does not have a name, now is the time to give it one. Silly, nerdy, and/or humorous names are especially appreciated.

Q2. (10 points) ..... Git Branch

Please commit your work to a new branch called `mostly-working` in your existing project repository.

Q3. (10 points) ..... Organization

Be sure to organize your implementation across at least three files, as in the previous assignment:

- Your parser should reside in a file called `ProjectParser.fs`.
- Your interpreter / evaluator should reside in a file called `ProjectInterpreter.fs`.
- Your `main` function, as well as any necessary driver code, should reside in a file called `Program.fs`.
- You may create additional library files as necessary.

All of these files should be stored somewhere in a directory called `lang`. To incorporate tests, you will need to reorganize your code as a .NET solution instead of a simple .NET console project. However, the precise arrangement of files inside the `lang` folder does not matter to me, and is totally up to you.

Your project implementation should adhere to the following running convention. You should be able to `cd` into the `lang` 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.

Q4. (40 points) ..... Complete project specification

Your updated project specification as a LaTeX source file and pre-built PDF. Please call the LaTeX file `lang-spec.tex` and call the PDF `lang-spec.pdf`.

If you have not done so already, please merge your project proposal with your project specification document from Lab 9. The project specification should be a complete document that explains the purpose, motivation, and technical implementation details of your language. A sufficiently-motivated user should have all the information they need in order to write programs in your language using your documentation.

Most of the sections are the same as before; sections that require new text are marked with a bold **NEW**. Please be sure to have the following sections:

(a) Introduction

What problem does your language solve? Why does this problem need its own programming language?

(b) Design Principles

Languages can solve problems in many ways. What are the guiding aesthetic or technical principles that underpin its design?

(c) Examples

NEW. Provide three *working* example programs in your language. If any of the examples from your proposal do not yet work, please either extend the language to support them or replace them with working examples. Explain exactly how to run each example (e.g., `dotnet run "example-1.lang"`) and what the expected output should be (e.g., `2`).

(d) Language Concepts

What are the core concepts a user needs to understand in order to write programs? Think in terms of both "primitives" and "combining forms." What are the key ideas and how are they combined?

(e) Formal Syntax

NEW. Provide a formal syntax for all supported operations, written in Backus-Naur form. This documentation should provide all of the rules necessary for a user to generate a valid program. You may omit whitespace from your BNF specification if you find it cumbersome to write about.

(f) Semantics

NEW. Update the semantics section from Lab 9 to explain all of your currently-supported data types and operations. This section should explain how a user understands the effect of a syntactic construct given in the formal syntax section. This need not be so detailed that it explains what the code *does*; instead it should explain what the syntax *means*. In other words, focus on *what* each language element achieves instead of explaining *how* it does it. Please refer to the example shown in Lab 9 for guidance. Your semantics section need not be in the tabular form shown if a table is inconvenient.

(g) Remaining Work

NEW. Add a section at the end of your specification that explains which features are not yet implemented but which you plan to implement by the final project deadline. This should include any essential remaining data types and operations described in your proposal that you have not yet implemented.

If you are already nearly done, this would be a good place to describe an optional "stretch goal." For example, if you plan to build a graphical user interface for your language—which is most definitely optional—describe that interface here. Another possibility is a program correctness checker. For example, your syntax may generously admit programs that make little sense syntactically; adding a program checking phase before evaluation is another good "stretch goal." A third possibility may be to describe a plan to enhance the readability of your language specification.

Q5. (10 points) ..... Example programs

Provide the example programs discussed in your proposal as separate files so that it is easy to find and use them. Please call them `example-1.<whatever>`, `example-2.<whatever>`, and

`example-2.<whatever>`. For example, I might call my example programs `example-1.lang`, `example-2.lang`, and `example-3.lang`.

Q6. (10 points) ..... Execution

Each of your examples 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.

Q7. (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. 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 previous functionality. But 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`. Once you have additional tests, I should be able to run those the same way.