Homework 10

Due Thursday, Nov 29 by 10pm

Project 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`.

For full credit, your code should both build and run correctly.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw9_<your user name>`. For example, my repository would be `cs334hw9_dbarowy`. Since this repository already exists from a previous assignment, please commit your work to a branch called `mostly-working`.

A good way to check that you submitted your assignment correctly is to `git clone` your repository to a new folder and then try building/running everything.

Pair Programming and Honor Code: You may optionally collaborate with one other person on the submission for this assignment. If you work with a partner, choose one of the two partner repositories for your submission. In the other repository, be sure to leave a `collaborators.txt` file that states who you worked with and the name of the repository where the code may be found. If you would like me to pair you with a partner, please let me know. You may not share code with any student who is not your designated partner.

This assignment is due on Thursday, Nov 29 by 10pm.

Guidelines

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 `hw9` repository. Additionally, if your language was stored before in a directory called `q1`, please rename it `lang`.

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`. The precise arrangement of files inside the `lang` folder does not matter; for example, you may decide to organize your implementation as a .NET solution instead of a simple .NET console project so that you can use MsTest.

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 Assignment 9. The projet 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.

(f) Semantics

NEW. Update the semantics section from Assignment 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 Assignment 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. (*0 points*) .... Tests

This submission does not require tests, however your final project will require unit tests that demonstrate that the example programs behave as expected. If you want to get a head start, you can work on them now.

Unit tests are especially useful for testing parsers and evaluators, which are likely to be pure functions in your language implementation. 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 do so, you will first need to `prepare` your input string, then pass it to one of your parser functions, then it will need to check for `Success` or `Failure`. Having parser tests makes the development of a language parser much easier because you can see whether the addition of new language syntax has caused problems recognizing inputs that you have been able to successfully parse before.