Lab 10

Due Sunday, Dec 3 by 10pm

Handout 26
CSCI 334: Fall 2023

Turn-In Instructions

Before starting work, create a new branch in your existing project-specific repository called `specification`. Commit and push work for this lab to the `specification` branch.

This assignment is due on Sunday, Dec 3 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 pair programming lab. Like previous partner labs, you may work with a partner. Unlike previous labs, you may collaborate to produce a single solution. You do not need to submit a `collaborators.txt` file for this lab.

Reading

1. **(Read)** Read "Appendix B: Branching in `git`" from the course packet.
2. **(As needed)** Read "Implementing Scope" and "Implementing Functions" from the course packet if your language makes use of these features.

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) .... 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 `Parser.fs`.
- Your interpreter / evaluator should reside in a file called `Evaluator.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 in a directory called `code`. The precise arrangement of other files inside the `code` folder does not matter to me, and is up to you.

For this checkpoint, your implementation does not need to do anything new beyond what was required in your minimal working version. I will check to see that you committed something, but I will not test it, so you are encouraged to continue hacking on it, or even leave it in a broken state for this checkpoint.

Q3. (60 points) .... Draft project specification

In this lab, you will create a draft project specification. As we're just getting started, your specification does not yet need to fully describe your final project. However, it should be "section-complete," meaning that your document should include text for all of the sections outlined below. When you submit your final project, at the end of the semester, your specification will need to fully describe your project.

Include your project specification as a LaTeX source file and pre-built PDF. Please call the LaTeX file `specification.tex` and call the PDF `specification.pdf`.

You may start by reusing the text you wrote for your project proposal. The project specification should explain the purpose, motivation, and technical implementation details of your language. By the end of the semester, a sufficiently-motivated user in possession of your specification should have all the information they need in order to write programs in your language using your documentation. For some of the sections below, you may not need to change your text much; for other sections, expect to write new text.

Please be sure to have the following sections:

(a) Introduction (≥ 2 paragraph)

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

(b) Design Principles (≥ 1 paragraph)

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

(c) Examples (≥ 3 examples)

Provide three example programs in your language that *will eventually work*. Unlike the previous text you wrote here, these examples should conform to your formal grammar. Explain exactly how each example will be executed (e.g., `dotnet run "example-1.lang"`) and provide the expected output (e.g., `2`).

(d) Language Concepts (≥ 2 paragraphs)

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 (as much space as needed)

Provide a formal syntax your language, 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 include.

Minimally, your BNF should include everything you have currently implemented in your small language. However, you are encouraged to add BNF syntax for features that you have not yet implemented, as a way of "thinking them through." The final version of this section should match your actual implementation, since I will be using it to understand your language and to write programs of my own.

(f) Semantics (1 short description per syntactic element)

If necessary, update the semantics section from your previous checkpoint 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. Your semantics section need not be in a tabular form if a table is inconvenient.

(g) Remaining Work (≥ 1 paragraph)

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 section should include any essential remaining data types and operations described in your proposal that you have not yet implemented. You can discuss remaining work informally; think of this section as a personal checklist.

Q4. (20 points) .... Example programs

Provide the example programs discussed in your Examples section 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`.