MiniML Extension Writeup

Cory Zimmerman, 2023

I extended MiniML by (1) building a lexical evaluator, (2) adding a `String` atomic data type, and (3) refactoring the project's core logic.

Lexical evaluation As recommended by the assignment guide, I added a third evaluation function obeying the lexical semantic model. Doing so turned out to be an excellent decision, as it pushed me to rewrite the Env.extend method to preserve physical equality and uncovered a need to more carefully manage bound variable names in function definitions for the dynamic and lexical evaluators.

Case-specific decisions are documented in the `eval_disinct` module. Ultimately, this evaluator correctly exhibits all the behavior I could think to test, and I'd consider it the most robust of my three evaluation implementations.

String data type I made this small extension after refactoring the project as described below, so it was incredibly quick to do. After extending the parsing logic in the .mll/.mly files, it required fewer than two lines of code to add string evaluation and string comparison to all three evaluators. Relying on the abstractions outlined below, the process was fast, easy, and worked as expected on the first try.

Design and style I pursued a heavy refactor of the MiniML project that meaningfully improved the modularity, readability, testability, and extensibility of my code. While the extension examples in the assignment guide were all feature-based, I chose to explore and apply the design and style principles of the course by improving existing logic. Because doing so took such a long time, I'm considering this refactor my main extension.

The refactoring process approximately followed this flow:

- Pulled out the Env module and signature into its own file, extended it with related definitions and type: `env.ml`
- Defined functions to handle the evaluation needs of each data type for each evaluator.
- Analyzed which methods were the same for all three evaluators, combined those methods so each evaluator called the same helper where possible.
- Created a common module with helper functions shared by the three evaluators: `eval_common.ml`
- Defined a module signature and created three modules with model-specific functions for each evaluator: `eval_distinct.ml`
- Packaged each model-specific evaluator's functions into a record that could easily be passed as a parameter.
- Created an abstract generic evaluation function combining the common evaluation logic with model-specific helpers passed as parameters. Connected the three evaluators in `evaluation.ml`.
- Added the `eval_utils.ml` module for helpers that weren't evaluation-specific.
- Combed through each file standardizing variable names, cleaning up logical patterns, fixing circular/ambiguous imports, and adding documentation where appropriate.

The process was quite intense, but I'm super pleased with how it turned out. As mentioned above, adding an additional atomic data type and binop comparison case was

almost effortless, and I see the project as much more readable and maintainable after these changes.

Speaking directly to the lessons of the course, the refactor required abstracting the common and distinct elements of the evaluators, decomposing the evaluation logic into distinct modules, and compartmentalizing methods by defining clear module signatures all while thinking carefully about the understandability and safety of my code.