Extension Writeup: Lexical Scoping

The extension chosen was the implementation of an evaluation function that evaluated expressions using lexical scoping. Lexical scoping involves evaluating expressions based on the state of the environment when they are declared. This is different from dynamical scoping, where expressions are evaluated based on the state of the environment at the time the expression is applied. For example, take the following expression:

```
let x = 2 in
let rec f = fun y -> x + y in
let x = 5 in
f 1;;
```

In a dynamically scoped environment, the value of `x` at the time of the application `f 1` is `5` thus the function f would evaluate to `fun y -> 5 + y`, making the final result of the evaluation `6`.

In a lexically scoped environment, the value of `x` is used from the time the function `f` is declared. So, the value of `x` within the function is `2`, which would make the function evaluate to `fun y -> 2 + y`, making the final result of the evaluation `3`.

I chose to extend the interpreter to evaluate expressions using lexical scoping because the full OCaml language evaluates expressions by scoping lexically. Making the MiniML interpreter use lexical scoping would thus make it more like OCaml in terms of accuracy.

The lexically-scoped evaluation was done in the function `eval_l`. The logic behind evaluating most types of expressions in `eval_l` was identical to that of the dynamically-scoped evaluation function `eval_d`. Much of the code from `eval_l` is very similar to that of `eval_d`. The differences are as follows:

- `Fun` expressions evaluate to a `Closure` type, which contains a pairing of the function itself and the environment at the time that the function was declared.This allows the evaluator to evaluate the function later using the state of the environment at the time of the function's declaration.
- `Letrec` expressions are evaluated differently. As per the project specification, a `Letrec` must originally extend the current environment to include the variable being defined with an `Unspecified` placeholder. Then the definition must be evaluated. Then, the environment is extended again to the variable and its now evaluated definition. After that, the `Unspecified` placeholder is replaced with the evaluated definition (using mutability). Finally, the body of the `Letrec` can be evaluated using the now twice-extended environment.
- `App` expressions must be handled differently, because the first expressionsof an `App` must now be a `Closure` rather than a `Fun`

The function `eval_l` took a good amount of time to debug. It does use some repeated code. But, because of the amount of effort that went into debugging and the type differenced between `eval_d`, I chose not to make a merged evaluation function that handled both lexical and dynamical scoping. I was afraid that the process of merging the two would create unanticipated bugs, which, given the time restraint, could have caused problems with submitting. I played it safe and kept the two functions separate.