Writeup: Lexical Scoping

In order to extend the scope of the miniml project I implimented an evaluation function that uses the environment model of evaluation yet also uses a lexically based scope similarly to Ocaml and the substitution based model of evaluation.

Given the function

```
let x = 1 in
let f = fun y -> x + y in
let x = 2 in f 3 ;;
```

the Ocaml interpreter as well as the substitution based evaluator would return a value of 4 despite that x is later bound to 2 making an evaluation to 5 also logical. However, the environment based evaluation function eval_d would indeed return 5. This is because the environment based evaluator is dynamically scoped and thus variables are bound based on the order that they are evaluated in, the latest being the binding that sticks. For my extension I created a lexically scoped evaluator that uses the environment model of evaluation.

Implementing a lexically scoped evaluator based on the environment model

Implementing this evaluator should have been quite simple in theory, hoewever, I wrote my eval_d function such that it wrapped a recursive helper function that returned values of type expr. In order to create eval_l, my lexically scoped evaluator, I copied eval_d and rewrote it so that the helper function returned values of type Env.val. I then changed the evaluation of functions so that instead of just returning the function back, it returned a closure containing the function and the environment a the time the function was evaluated. This is crucial as I will explain later.

Next I changed the evaluation of apps so that instead of requiring the first expression of the app to be a function, it would need to be a closure containing a function and an environment. I then evaluated the app just as I evaluated the app in eval_d except instead of using the current environemnt extended by the second expression of the application to evaluate the expression of the function, I used to old environment that was returned in the closure. This meant that the function application evaluates in the lexical scope evaluating variables to the value that they contained when the function is originally evaluated.

Finally, I had to implement the way eval_l evaluated recursive functions so the information about the old environment is preserved. In my eval_l function there are several times where I use a helper function to extract the expression from arguments of type Env.val to make the easier to work with. However, this means that in the case where some bindings are nested, the closure of the function may be erased and later replaced with Env.Val, losing the old environment and causing the application evaluation to break. In order to fix this instead of extending the

environment in my evaluation of let rec to change unassigend to the definition, I simply update the reference. This preserves all of the information and allows eval_l to work in cases of nested bindings.