Extension Writeup - Lexical Scoping

The extension that I implemented was the lexically scoped evaluator. In order to implement this, I used much of the code from my eval_d (dynamically scoped evaluator) function. I essentially rewrote the entire eval_d function with Closures substituted in for Vals as the return values and then modified my function application implementation. For example, if I ran into a Num n, instead of returning that expression wrapped as a Val, I returned it wrapped as a Closure with the environment (env) that had been passed through the evaluator function recursively. I wrapped up the expression and environment as a closure using the Env.close function.

```
match exp with
  | Num _ | Bool _ -> Env.close exp env
  | Var id -> Env.lookup env id
  ...
```

I modified my function application implementation for the lexical environment by following the guidance given in the final project PDF. For function application, I extend the environment contained in the Closure of the evaluation of the body to include the evaluation of what the function is applied to in the overarching environment. This supports multiple function applications, for example, f (f (f 5)).

```
| App (e1, e2) ->
    (match eval_l e1 env with
     | Val _ -> raise (EvalError "unclosed environment")
     | Closure (e', env') ->
       (match e' with
        | Fun (id, e) ->
          eval_l e (Env.extend env' id (ref ((eval_l e2 env))))
        | _ ->
          raise (EvalError "argument is not a function - cannot be applied")))
```

One aspect of my match statements in eval_l that is different from my match statements in eval_d is how I separate the Val and Closure match. In eval_d, I only want to evaluate inside of the overarching environment. I never deal with a Closure. So, whether I run into a Val or Closure does not matter. I only want to extract an expression. Thus I match as follows:

```
(match eval_l e1 env with
  | Val e | Closure (e, _) ->
   (match e with ...
```

In my match statements in eval_l, I separate the Val and Closure statements because in many cases I want to extract the current environment as well as the expression to pass both into a recursive call. I cannot match Val and Closure in the same statement because the environment within the Closure that I recursively pass would need to be present on both sides of the match statement. Also, I know that I will never run into a Val statement while evaluating eval_l, so if I somehow run into a

Val statement I simply raise an evaluation error. The only times I don't need to worry about the closure environment are when I run into binary or unary operators. The only expressions related to these operators can be Nums because all other expressions should have already been bound and evaluated to their values. So, after performing these operations on their expressions, there is no new binding that I have to create. Thus, I can just close them within the overarching environment. I match these as follows:

```
  | Unop (_op, e) ->
    (match eval_l e env with
     | Val e' | Closure (e', _) ->
       (match e' with
        | Num n -> Env.close (Num (-n)) env
        | Bool b -> raise (EvalError ("can't negate bool " ^ string_of_bool b))
        | Var id -> raise (EvalError ("unbound variable " ^ id))
        | _ -> raise (EvalError "cannot be negated")))
  | Binop (op, e1, e2) ->
    (match eval_l e1 env, eval_l e2 env with
      | Closure (e1', _), Closure (e2', _) ->
        (match e1', e2' with
        | Num n, Num m -> (match op with
                           | Plus -> Env.close (Num (n + m)) env
                           | Minus -> Env.close (Num (n - m)) env
                           | Times -> Env.close (Num (n * m)) env
                           | Equals -> Env.close (Bool (n = m)) env
                           | LessThan -> Env.close (Bool (n < m)) env)
        | Var id, _ -> raise (EvalError ("unbound variable " ^ id))
        | Bool a, Bool b -> (match op with
                             | Plus | Minus | Times -> raise (EvalError
                              (exp_to_string exp ^ " wrong type for operation"))
                             | Equals -> Env.close (Bool (a = b)) env
                             | LessThan -> Env.close (Bool (a < b)) env)
        | _ -> raise (EvalError (exp_to_string exp ^ " error")))
      | _ -> raise (EvalError "unclosed environment"))
```

In testing, I was able to test my lexical scoping and substitution evaluator using the same evaluation test function. Since lexical scoping and substitution give the same results, I wrote my testing function such that I could pass in any evaluator. I also wrote a function v2e, which converts a value to an expression. I used this in my testing function because substitution returns Vals while lexical scoping returns Closures.

```
let v2e (v : Env.value) : Expr.expr =
  match v with
  | Env.Val e | Env.Closure (e, _) -> e
;;

let test_eval_sl (eval : Expr.expr -> Env.env -> Env.value) () =
  ...
;;
```