For this project, I implemented 4 extensions:

1. Implemented *eval_l* without stores (Figure 19.2 of textbook). Edited miniml.ml to print the output from all three evaluators on one user input along with the abstract syntax of that input.

2. Eliminated redundant code within evaluators (Design).

3. Added floating point arithmetic functionality through updates of expr.ml, evaluators, miniml_lex.mll, and miniml_parse.mly.

4. Added string concatenation functionality through updates of expr.ml, evaluators, miniml_lex.mll, and miniml_parse.mly.

To illustrate those extensions, I'll share screen grabs from my implemented interpreter (items 1, 3, and 4) and screen grabs of code (item 2). Below are those images and accompanying description.

1. See item 3.

2. To eliminate redundant code I created an ADT, *eval_type*

```
type eval_type =
  | Eval_s
  | Eval_d
  | Eval_l
```

   I then added this ADT as an optional arg to *eval_s*. Within *eval_s*, *eval_type* is used to determine which evaluator is used on expressions found in match constructs whose effect is the same between all three evaluators.

```
let rec eval_s ?(eval_type : eval_type = Eval_s)
               (exp_s : expr)
               (env_s : Env.env) : Env.value =
  let f =
      match eval_type with
      | Eval_s -> eval_s ~eval_type:Eval_s
      | Eval_d -> eval_d
      | Eval_l -> eval_l in
```

   The effect is that *eval_d* and *eval_l* have redundant code eliminated through an appropriate call to *eval_s*.

```
and eval_d (exp_d : expr) (env_d : Env.env) : Env.value =
  match exp_d with
  | Let (x, def, body)
  | Letrec (x, def, body) -> (
    match eval_d def env_d with
    | Env.Val vD -> eval_d body (Env.extend env_d x (ref (Env.Val vD)))
    | _ -> raise (EvalError "Let: dynamic env. semantics don't use Closures"))
  | App (e1, e2) -> (
    match eval_d e1 env_d with
    | Env.Val (Fun (x, body)) ->
      eval_d body (Env.extend env_d x (ref (eval_d e2 env_d)))
    | _ -> raise (EvalError "App: first expression must be a function"))
  | Var x -> Env.lookup env_d x
  | _     -> eval_s ~eval_type:Eval_d exp_d env_d
```

```
and eval_l (exp_l : expr) (env_l : Env.env) : Env.value =
  match exp_l with
  | Let (x, def, body) ->
    let vD = eval_l def env_l in
    eval_l body (Env.extend env_l x (ref vD))
  | Letrec (x, def, body) ->
    let vD = ref (Env.Val Unassigned) in
    let env_x = Env.extend env_l x vD in
    vD := eval_l def env_x;
    eval_l body env_x
  | App (e1, e2) -> (
    match eval_l e1 env_l , eval_l e2 env_l with
    | Env.Closure (Fun (x, body), env_l) , vQ ->
      eval_l body (Env.extend env_l x (ref vQ))
    | _   -> raise (EvalError "App: first expression must be a function"))
  | Var x -> Env.lookup env_l x
  | Fun _ -> Env.close exp_l env_l
  | _     -> eval_s ~eval_type:Eval_l exp_l env_l ;;
```

3. Below I show showcase some functionality for floating points along with where the lexical and dynamic evaluators differ:

```
<== let rec f = fun x -> if x = 0. then 1. else x *. f (x -. 1.) in f 4. ;;
--> Letrec("f", Fun("x", Conditional(Binop(Equals, Var "x", Float 0.), Float 1., Binop(F_Times, Var "x", App(Var "f", Binop(F_Minus, Var "x", Float 1.))))), App(Var "f", Float 4.))
s==> 24.
d==> 24.
l==> 24.
<== let x = 21. in let f = fun y -> x *. y in let x = 2. in f x ;;
--> Let("x", Float 21., Let("f", Fun("y", Binop(F_Times, Var "x", Var "y")), Let("x", Float 2., App(Var "f", Var "x"))))
s==> 42.
d==> 4.
l==> 42.
<==
```

Lastly, here I show how my edits to miniml_ml effect what is printed. Notice how the *value* returned by evaluating a *Fun* includes the expected *Closure* characteristic of lexical evaluation:

```
<== fun x -> x ;;
--> Fun("x", Var "x")
s==> fun x -> x
d==> fun x -> x
l==> [{} ⊢ fun x -> x]
<==
```

4. Below I show a few ways in which you could say hello to the world:

```
<== let x = "!" in "Hello world" ^ x ;;
--> Let("x", Str "!", Binop(Concat, Str "Hello world", Var "x"))
s==> Hello world!
d==> Hello world!
l==> Hello world!
<== (fun x -> "Hello world" ^ x) "!" ;;
--> App(Fun("x", Binop(Concat, Str "Hello world", Var "x")), Str "!")
s==> Hello world!
d==> Hello world!
l==> Hello world!
<==
```

As a side note, I also added division functionality for both integers and floating points as well as a greater than binary operator. These seem less significant than the rest so I leave them here for completeness only.

```
<== 2. /. 3. ;;
--> Binop(F_DividedBy, Float 2., Float 3.)
s==> 0.666666666667
d==> 0.666666666667
l==> 0.666666666667
<== 2 / 3 ;;
--> Binop(DividedBy, Num 2, Num 3)
s==> 0
d==> 0
l==> 0
<== 2 > 3 ;;
--> Binop(GreaterThan, Num 2, Num 3)
s==> false
d==> false
l==> false
<==
```

This project was a blast. My only regret is not spending more time on it in the beginning. You all are wonderful and I appreciate your abundance of patience and help this semester.