For this project, my extensions involved implementing an evaluator for lexical semantics, creating a general evaluator function to abstract away commonalities between eval_l and eval_d, and adding support for exceptions. This writeup serves to detail the implementation, execution, and purpose underlying each extension in order to best enhance the functions of the MiniML interpreter.

1) eval_l: Lexical Semantics Evaluator

This evaluating function is largely similar to eval_d, but it relies on the use of closures to evaluate a function at the environment in which it was defined as opposed to when it was called. The main differences in implementation lie in the Fun and App match cases.

Evaluating a function simply returns a closure of the function and the current environment it was defined in, utilizing the Closure construct of Env.Value. As for evaluating an application, I first evaluated the function in order to access the resulting closure in a match statement. I then extended the environment to map the variable in the function to the evaluated argument to which the function is being applied. This then allowed me to evaluate the body of the function in this new environment, which preserves the value of the function at the time it was defined and thus rules the structure of lexical semantics. I realized that evaluating a let rec function in lexical semantics is largely similar to doing so in dynamic environmental semantics: I temporarily assigned a temp variable to an Unassigned ref, which allowed me to extend the environment to map to the temp variable. By mutably changing temp to be the evaluated definition in this extended environment, I could properly evaluate the body of the function in this new extended environment while accounting for the recursion.

To provide an example of the main difference between the dynamic implementation and the lexical implementation, we use this function:

```
    let x = 2 in let f = fun y -> x + y in let x = 8 in f x ;;
```

In dynamic environment semantics, this function would evaluate to 16 since x = 8 at the time the function is called. This then makes the function be (8+8). For lexical environment semantics, however, the function would evaluate to 10. A closure preserves the function definition to be fun y -> 2 + y since x = 2 at that time. Thus, when the function is called with x = 8 as an argument, the function instead evaluates to (8+2).

2) eval_h: Evaluator Helper Function

Considering that much of the functionality between eval_d and eval_l were the same, I decided to create a generalized eval function to abstract away the commonalities. This function, in addition to an expression and an environment, takes in an eval function as an argument so that it can call a specific eval function to execute cases that are unique to that evaluator. For instance, while evaluating variables and let statements are the same in both eval_d and eval_l, the two functions differ in how they evaluate a function. In this case, eval_h calls "eval exp env" so that when eval_h is called to evaluate a let statement in eval_d, the function within the let statement can still be evaluated dynamically, despite the fact that a let statement syntactically is the same for eval_d and eval_l.

I decided to create a helper function for eval_d and eval_l as opposed to one that abstracts away similarities between those and eval_s because I prioritized making eval_d and eval_l as efficient as possible. Since there are more similarities between these two functions than there are between eval_s, eval_d, and eval_l, I found it more worthwhile in design to minimize as much redundancy in my code as I could.

3) RaiseExn: Exception Support
I decided to extend the scope of exceptions because I found them to be too generalized to account for the various cases of error handling. I realized that rather than using EvalError for errors that widely differed, I could create another exception that handled a more specific yet frequently recurring case. I added the RaiseExn construct to the Expr type and included other changes in miniml_lex.mll, miniml_parse.mly, and expr.mli so that the parser could register and properly interpret the new construct. I also added the additional exceptions to miniml.ml to generate error messages when these cases appeared.

Specifically, RaiseExn raises "ImpossibleCase" to account for match cases that are never supposed to arise. For instance, in the evaluators, RaiseExn is returned to handle the case when the first expression in App is not a function as it should be. I also created an exception with a string argument called TypeError that is raised when functions are applied to inappropriate types. This is seen in my helper functions, unopeval and binopeval, to address match cases that attempt to execute arithmetic operations on non-integer arguments.

My primary goal for these extensions was to exemplify the theme of abstraction and efficiency in my code. While I could have introduced additional features to the interpreter such as more operator constructs or atomic types, I chose to improve upon the given features of the interpreter so that its limited functionality could be as effective as possible. I found it more rewarding to make my extensions improve the quality of the interpreter as opposed to strengthening its quantity of features. Ultimately, the themes of design and abstraction emphasized throughout the semester are what motivated my design decisions.