Commentary on my Extensions

I implemented two extensions to my Miniml evaluator, the first of which being that the introduction of units to the expressions type. This was a relatively minor extension, as it came about after a failed attempt to implement refs into my evaluator (which of course would have required a unit type for the outputs of the unops "ref" and "!"). It was implemented through adding it to the expressions type in both expr.ml and expr.mli, adding it as a token in miniml_parse.mly, and giving it both concrete and abstract string versions of itself. It doesn't add much to the evaluator as it is a very basic type; the sole somewhat practical use of it that I could implement was found in comparisons, where a unit would always be equal to another unit and would never be less than another unit. Since a unit doesn't have any free variables, or any variables for that manner, it can only evaluate to itself. Still though, it's a necessary component of OCaml so I felt it was pertinent to keep it in the evaluator. To use it, you have to actually write "unit", as representing it as "()" would confuse the evaluator which already takes parentheses as its opening and closing symbols. (See Image 1 in link on bottom)

Secondly, I implemented eval_l, an evaluator that, like OCaml, assesses code lexically. It was implemented by making much of the code of eval_d, a dynamic evaluator, global so that it could be shared between the functions that are very similar. eval_l takes in an expression and an environment and returns a value, and mainly differs from eval_d in their respective implementations of the Fun and App subtypes of type expressions. While eval_d simply returns a Fun as a value version of itself and evaluates App(e1, e2), in the case that e1 can be evaluated in the environment to a value of a Fun, by evaluating the body of the Fun in an environment formed by the extension of the current environment to include a binding of the Fun's head variable to vq, the value obtained by evaluating e2 in the current environment. Meanwhile, eval_l evaluates functions to closures that contain both itself and the environment in which this evaluation is performed, while application App(e1, e2), again in the case that the evaluation of e1 results in a value (in this case a closure, in contrast to the usual Val) of a Fun (it should be noted that in the case that this prerequisite is not met both eval_d and eval_l return errors, another similarity between the two), after again calculating the value of vq using e2 and the current environment, evaluates the body of the function in a new environment formed by extending the current environment to including the environment found in the closure, the environment in which the Fun was initially evaluated in and from, the environment in which the binding of the head variable of the Fun to vq occurred. With these changes, my evaluator can now, instead of evaluating in the most recently encountered environment, evaluate in the top level environment, the one where a function was first called, just like how code is evaluated in OCaml.

[Images](https://docs.google.com/document/d/1v3RvhVS-N0qm6Vl9p-2gl5ktDGBbds2l1F4YWomuwSw/edit?usp=sharing)