Final Project Write-Up

The first extension that I implemented was the lexical environment semantics as the process made sense after completing eval_d. I followed much of the same reasoning from the other eval functions while also relying on the textbook for the exact rules.

After that, I implemented more operators for the interpreter. Among the binop operators, I noticed that division and the greater than operator were missing. Inserting them into the main functions wasn't overly difficult as the followed the same rules as multiplication/ the greater than operator. More challenges came when I tried to become familliar with the parser to update the operators. It was a fun process to learn about parsing and go through trial and error, but I followed the clues that were present to make them work.

For additional unary operators, I added more math functions, Square Root and Log. These functions accompanied the addition of floats into the expression types because they will only work when applied to floats. I also added strings to the type list. At first, the interpreter viewed all strings as variables and I had to learn how to change that in the parser.

I also added refs into the interpreter, which was a made a little easier due to the elements of mutability in the project. There were three parts to the ref extension. The ref type is attached to another expression and can take in every expression type. I added de-referencing as a unary operator which was straightforward. After ensuring that the argument is a ref, I was able to just return the value stored within the ref. So, abstractly, a deref would look like: Unop(Deref, Ref(Num 5)) and it would evaluate to Num 5. Finally, there is the binary operator which I called RefAlter which references ":=". This was slightly more work as I had to ensure the first expression is a ref before dereferencing it. Then, I used a helper function that compared the types of two expressions and determined if they were the same. If the two expressions were the same type the value within the ref is swapped. So, if you attempted to evaluate Binop(RefAlter, Ref(Num 2), Bool true), there would be an error. On the other hand, evaluating Binop(RefAlter, Ref(Num 2), Num 8) would result in Ref(Num 8). By this point, I was more familiar with the parser, so the process of introducing it to the interpreter was smoother.

One failed extension that I tried was implementing lists, but I wasn't able to alter the parser to recognize every type of list consistently. However, I was able to implement the relevant code in expr and evaluation.