CS51 Final Project Extension Writeup

For my extension to this project I decided to implement refs in MiniML. As suggested by the project, I implemented these three functions into the language:

```
ref : 'a -> 'a ref
(!) : 'a ref -> 'a
(:=) : 'a ref -> 'a -> 'a ref
```

Note that (:=) does not return a unit, as it generally does in OCaml – it instead returns the new reference after it has been assigned. I was trying to challenge myself to do this without implementing units or the execution of multiple statements separated by single semicolons.

Putting the necessary code into the lexer and parser was pretty simple and actually a lot of fun. The actual process of figuring how to implement these tokens' underlying semantics was a bit harder. I eventually realized that things were a bit simpler than I thought at the beginning. A MiniML ref was just a gloss over an OCaml reference to a MiniML expression, and (!) and (:=) were just Unop and Binop abstractions of these respective operators in OCaml. This allowed for the enabling of the classic pointer-reference trick where two variables' values are changed with one statement:

```
let a = ref true in let b = a in let c = (a := false) in !b
;;        (* false *)
```

A way in which my implementation of refs differs from proper OCaml is that I treat 'a refs as first-class values, the evaluation of whose interior can be delayed and eventually returned with the bang operator. This may seem like a somewhat unorthodox choice, but it parallels the differences between OCaml and MiniML when it comes to functions. OCaml's functions evaluate and are checked for errors when they are first declared; MiniML waits for an application of a function before it tests the function for errors. This can lead to some rather unorthodox, but understandable behavior such as:

```
let rec x = ref x in !x;;          (* ref x *)
```

OCaml's wonderful type inference deals with this nicely; my version of MiniML does not, and this is an obvious future direction toward which more development could focus.

One other slight extension I made to the language as I was working on it was operator type-checking. I made sure that each Unop and Binop could only operate on the types of expressions that it made sense on which to operate: (+) on Nums, (=) on Bools and Nums, and (!) on only Refs, for example. Particularly with the extended polymorphic comparison functions (=) and (<), this turned out to be somewhat of a pain, but it was eventually completed and here I am now. Any violation of the type invariants on operators results in an EvalError detailing something like “Error: Mutation only works on refs”, so that an end-user may quickly spot his/her error.