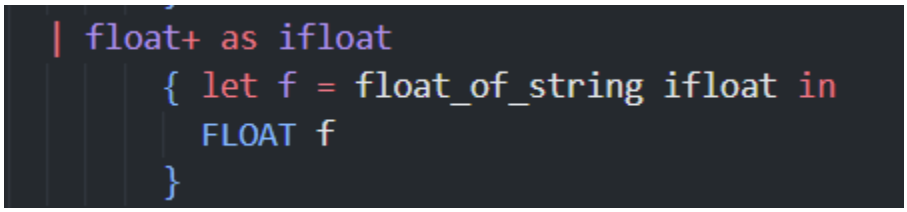
CS51 : Project Writeup

Type Extensions:

The atomic type extensions that I added were Floats and Strings. In order to implement the two types I needed to add them under the `expr` type in both `expr.ml` and `expr.mli`. In the `expr.mli` file they were added Float of float and String of string respectively. Additionally, changes needed to be made to the `miniml_lex.mll` and `miniml_parse.mly` files. To alter these files, I looked at how Num atomic type was implemented, and then based my implementation of Float and Strings off of that. In the `miniml_lex.mll` file I added

- `let float = digit* '.' digit*`

From this we have a float represented as a set of digits, a dot representing a decimal, and then another set of digits to follow accordingly. Along with this the following token was added to the file:

- 
```
| float+ as ifloat
    { let f = float_of_string ifloat in
      FLOAT f
    }
```

Here I made sure to follow a similar structure to the digit token in order to keep consistency between Nums and Floats. Then in the `miniml_parse.mly` all I needed to do was look for where INT was implemented, and then simply make a copy of the implementation except using FLOAT instead.

Then for strings I added the following into the `miniml_lex.mll`:

- `let string = '"' [^ '"']+ '"'`

As such, the string is represented as a set of zero or more characters that are not a " in between a set of quotes. From here I followed a very similar approach for implementing the float token by creating the string token as such:

- ```
  | string as istring
      { let str = String.sub istring 1 (String.length istring - 2) in
        STRING str
      }
  ```

Again, I tried to follow the structure of the digits token. I use String.sub in order to extract the actual string out from the parser, which allows me to just return the string in quotes. Finally, in `miniml_parse.mly` I again simply looked for where I saw the INT keyword and just tried to pattern match switching out INT for STRING where I saw fit.

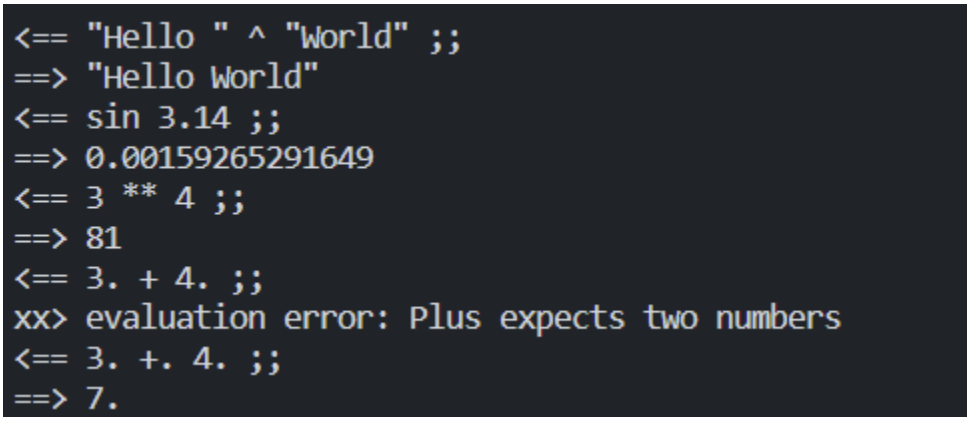
From here, to actually use these extensions I needed to add some operations. In Unop I added operations FloatNegate, Not, Sine, Cosine, Tangent, and the NaturalLog. In Binop I added PlusFloat, FloatMinus, FloatTimes, Divide, FloatDivide, Power, NotEquals, LessThanOrEqual, GreaterThan, GreaterThanOrEqual, and finally Concat. The actual implementation for these was not all that difficult. It involved mainly just extending the pattern matches that I already had, except now adding the operators for floats as well, along with their respective error codes (strings were only needed for Concat). Then when going into `miniml_lex.mll` and `miniml_parse.mly` it was slightly difficult to implement the new operations, however, with some pattern matching and looking for keywords similar to what I was adding as extensions (for example looking for LessThan when trying to add GreaterThan) made it easier. I needed to add tokens for all of them as well as their associativity as pictured below:
  ```

- ```
  %token EOF
  %token OPEN CLOSE
  %token LET DOT IN REC
  %token NEG FLOATNEGATE
  %token NOT
  %token SINE COSINE TANGENT
  %token NATURALLOG
  %token PLUS FLOATPLUS MINUS FLOATMINUS
  %token TIMES FLOATTIMES DIVIDE FLOATDIVIDE
  %token POWER
  %token EQUALS NOTEQUALS
  %token LESSTHAN LESSTHANOREQUALS GREATERTHAN GREATERTHANOREQUALS
  %token CONCAT
  %token IF THEN ELSE
  %token FUNCTION
  %token RAISE
  %token <string> ID
  %token <string> STRING
  %token <int> INT
  %token <float> FLOAT
  %token TRUE FALSE
  ```
- ```
  %nonassoc IF
  %left EQUALS NOTEQUALS
  %left LESSTHAN LESSTHANOREQUALS GREATERTHAN GREATERTHANOREQUALS
  %left PLUS FLOATPLUS MINUS FLOATMINUS
  %left TIMES FLOATTIMES DIVIDE FLOATDIVIDE
  %right POWER
  %right CONCAT
  %nonassoc NEG
  %nonassoc FLOATNEGATE
  %nonassoc NOT
  %nonassoc SINE COSINE TANGENT
  %nonassoc NATURALLOG
  ```

I also needed to deal with the grammar part, but that actually wasn't as hard, surprisingly, as I paid no mind to the numbers that were there and kind of just matched things up. At that point I had gotten used to it as it was the last edit I made to the two files, so I trusted that matching them up worked. Extending them into the actual eval functions was also not all that bad, as again it mainly involved just extending the pattern matches that I already had for the operations. I tried to stay consistent in my code, making sure to group things like comparisons together, and then group things like addition and subtraction together, and then also grouping times and divides together. It was especially important to edit the lex and parse files to add the keywords and

symbols that miniml would need to look for in order to perform the operations. For some of the operations, like sin, I needed to convert back and forth from ints and floats as the sin operations only take in floats. Finally, I made sure my error statements were consistent and that Nums and Floats didn't accidentally cross. This also extended to the power operation.

Here is some of the operations with their new types in action:

- 

Lexical Extension:

Along with the types, the lexical scope was also added to `evaluate.ml`. To implement this I followed the README as it provided good guidance. The only differences between the lexical and dynamic scoping were the Fun, Letrec, and App cases. For the Fun case, it was an easy change as I just needed to represent it as a closure instead of a Val. Then for Letrec I followed the readme and the rules from the book, creating a reference to an unassigned and then updating it from there. Finally for App, all that really needed to change was the format of the match cases, accounting for closures instead of vals. I then abstracted it by adding an overall evaluate function which handled both lexical and dynamic scoping based on the scoping type that I created. In the unit tests for evaluation, I made sure to have a couple cases where the two differed in evaluation,