For this sprint, Kwan dropped, and I was pretty busy, so the only thing we got done was the type checker. We didn't have time last sprint to do the type checker, and the type checker is basically the only thing after the parser that validates the correctness of the code, so there weren't any tests that checked invalid coder or types.

A quick note. The type of a type is a kind. * represents the kind of all types. For example, the types Int, and Maybe Int have a kind of *. The type constructor Maybe has a kind of * -> *, representing that it takes a type and gives a type. A more complicated type constructor like Either may have a kind of * -> * -> *. The haskell standard supports type constructors which takes other type constructors, which may have a kind that looks like * -> (* -> *) -> *.

Currently in THC, type variables may only have a kind of *. Attempting to use a type variable as anything that is not a * is undefined behaviour, as most of the code does not support it, but some of the code might because my mind wasn't very clear when implementing it. I should probably go through and make sure that any time a partial type constructor is encountered it just errors, but we will need to have type variables which support more complex kinds if we ever decide to implement monadic io.

All semantic analysis is done by the type checker. Type checking the data definitions is easy. The type checker first scans through and finds all the type names. The type checker also makes sure that every field in every constructor is a valid type. This is done by checking that the type constructor exists, the number of type arguments is equal to the number the type constructor expects, and recursively checking that each argument is a correct type. This is where the assumption where all type variables are * comes in. Some kind inference would have to be done if we allowed non * kinds as type variables, so we will just leave it alone for now.

Type checking is a bit more complicated for functions. First, the function type is checked to make sure it is well formed. Then, a dictionary is made of every given function argument and its type. The target type is the remaining right side of the function type which has not been assigned a type. Then the type of the body is inferred, and checked to see if it matches the expected output type.
To infer and check the type of the body, we recursively check and infer the function applications and cases which make up the body.
To infer the function body, the types of the function and the argument are inferred. We then attempt to coerce the types of the function and argument to match up with each other, where unbound type variables are bound and substituted in each type so the argument taken by the function and the argument given have the same type. Returned type of the function may also incur some substitutions.
For case statements, first the type of the expression which must be cased is inferred. Then, for each case statement, its type is evaluated, and any unresolved type variables in the expression may be inferred if needed. When inferring each case statement, some more variables are added to the dictionary containing the scope, which represent the variables matched to the case.

There is one exception when trying to coerce type variables, and it’s that type variables defined in the function type may not be coerced into another type. This is because if a function is defined as being general over all types, none of the arguments should need to be coerced into any particular type, and the output should also not be coerced into any particular type.

The tests work the same as before, you’ll have to visually inspect the output, since the actual type checker errors need to be reworked to be more friendly. There are more tests which check for invalid syntax and semantics now.