Script for Monad Transformers talk

\*greetings\*

Slide 1:

* talk is not meant to go in-depth into monad transformers and how they work
* aims to give a feel for the motivations behind monad transformers
* there are definitely people way more qualified to give this talk here, so do correct me if I state something glaringly wrong

slide 2:

* we’ll talk about what monad transformers are, the motivations behind them, and how to implement our own
* short exemplification of monad transformers in use within baby-l4/lsp

slide 3:

* “the interface”
* ~~Base Haskell itself does not include any notion of effects; it concerns itself only with evaluation/reduction of expressions into other expressions (instead of execution) e.g. 1 + 1 reduces to 2~~
* Monads such as IO, Except or Maybe allow us to include and execute semantically meaningful computations that extend beyond the evaluative capabilities of base Haskell.
* A try-except statement for handling errors is pre-baked into an impure language like python, but is handled by essentially the Except datatype within Haskell.
* A monad transformer, on the other hand, is a \*read from slide\*
* Essentially, a data structure containing two monads (one of them pre-defined) that shares the behavior of both monads. It’s called a monad transformer because it takes a monad as an argument, and returns a monad as a result.
* Why does this matter? Why can’t I just stack the monads on top of one another and call it a day?

Slide 4:

* The key idea here is that monads don’t play well with each other, in the sense that they don’t compose.
* The application of a composed functor consisting of two functors, f & g, is equivalent to the application of f, to the results of the application of g. (the composition of two functors returns a functor)
* But this isn’t necessarily the case for monads

Slide 5:

* \*talk through proof\* -> Since IO & Maybe semantically mean different things, they’re not interchangeable and a nested IO Maybe or Maybe IO might not necessarily follow the monadic laws
* you can’t use operations you’ve associated with monads like bind and join without knowing if they’re gonna misbehave.
* But inevitably, there are situations where we need more than one monad. So we’re going to have to learn how to handle this with monad transfomers

Slide 6:

* Firstly, we want to define the data structure we will be using. Monad transformers are defined wrt to their precursor monads, and we will be using the Maybe monad as an example here. \*describe data definition\*
* Afterwhich, because a monad transformer is supposed to be a monad, we want to write a concrete instance of the monad typeclass for our datatype.
* Some points to note:
  + ‘return’ in the return of the monadic transformer refers to the base monad’s instance of return
  + (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b

Slide 7:

* Now that we have some foundational knowledge behind monad transformers, we can begin to talk about how they are used in actuality
* For the LSP scope of work within baby-l4, there is a module called Language.LSP.Server. It contains the functions and types that deal with the running of the LSP server, which serves the diagnostics needed.
* \*