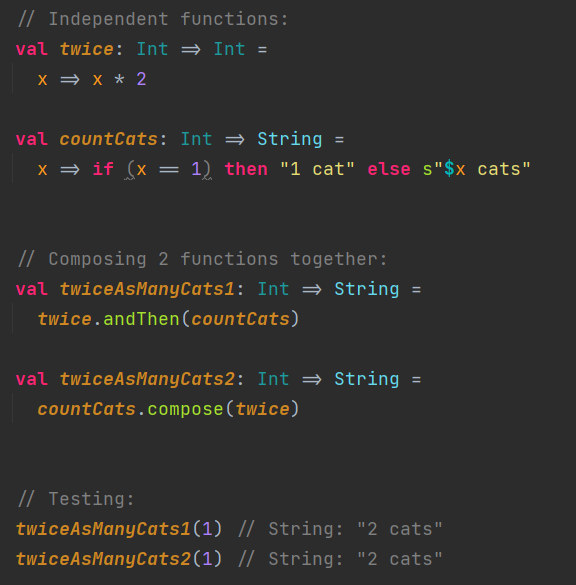
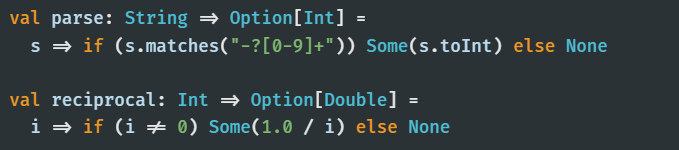
**Kleisli:**

Kleisli enables composition of functions that return a monadic value, for instance an Option[Int] or a Either[String, List[Double]], without having functions take an Option or Either as a parameter, which can be strange and unwieldy.  
  
We may also have several functions which depend on some environment and want a nice way to compose these functions to ensure they all receive the same environment.  
Or perhaps we have functions which depend on their own "local" configuration and all the configurations together make up a "global" application configuration.  
How do we have these functions play nice with each other despite each only knowing about their own local requirements?

**FUNCTIONS:**

One of the most useful properties of functions is that they compose. That is, given a function A => B and a function B => C, we can combine them to create a new function A => C. It is through this compositional property that we are able to write many small functions and compose them together to create a larger one that suits our needs.



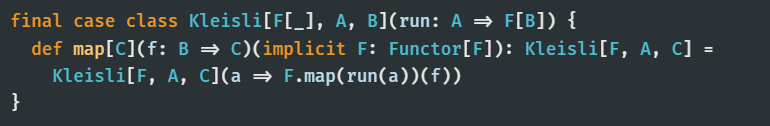
Sometimes, our functions will need to return monadic values…

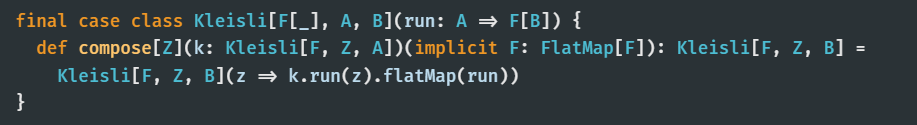
Can’t use Function1.compose (or Function1.andThen) to compose these two functions. The output type of parse is Option[Int] whereas the input type of reciprocal is Int.

This is where Kleisli comes into play.

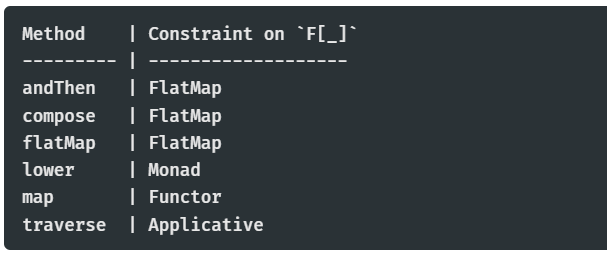
**STRUCTURE:**

Kleisli[F[\_], A, B] is just a wrapper around the function A => F[B].

When F[\_] is a Functor instance:  


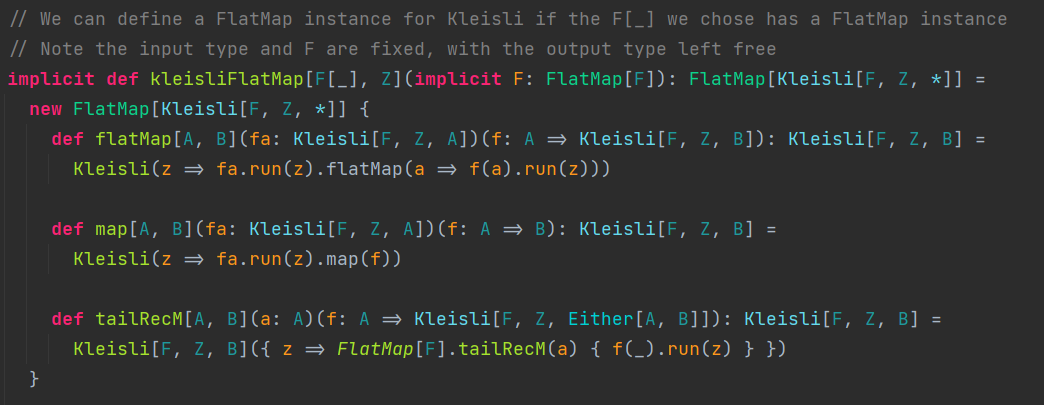
When F[\_] is a FlatMap instance:  


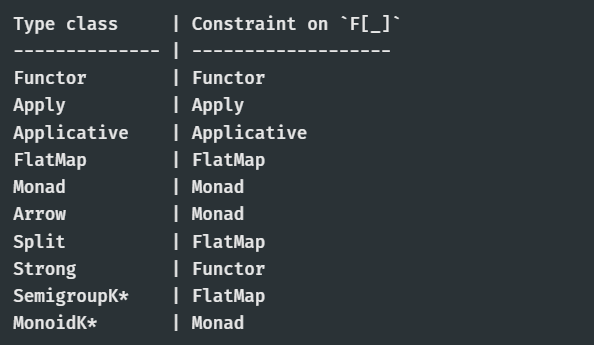
Depending on the properties of the F[\_], we can do different things with Kleislis.  
For instance, if F[\_] has a FlatMap[F] instance (we can call flatMap on F[A] values), we can compose two Kleislis much like we can two functions.

Below are some more methods on Kleisli that can be used as long as the constraint on F[\_] is satisfied:  


**TYPE CLASS INSTANCES:**

The type class instances for Kleisli, like that for functions, often fix the input type (and the F[\_]) and leave the output type free. What type class instances it has tends to depend on what instances the F[\_] has. For instance, Kleisli[F, A, B] has a Functor instance as long as the chosen F[\_] does. It has a Monad instance as long as the chosen F[\_] does. The instances in Cats are laid out in a way such that implicit resolution will pick up the most specific instance it can (depending on the F[\_]).

An example of a Monad instance for Kleisli…  


Below is a table of some of the type class instances Kleisli can have depending on what instances F[\_] has:  


\*These instances (SemigroupK & MonoidK) only exist for Kleisli arrows with identical input and output types: Kleisli[F, A, A] for some type A. These instances use Kleisli composition as the combine operation, and Monad.pure as the empty value.

Also, there is an instance of Monoid[Kleisli[F, A, B]] if there is an instance of Monoid[F[B]]. Monoid.combine here creates a new Kleisli arrow which takes an A value and feeds it into each of the combined Kleisli arrows, which together return two F[B] values. Then, they are combined into one using the Monoid[F[B]] instance.

**Other uses**

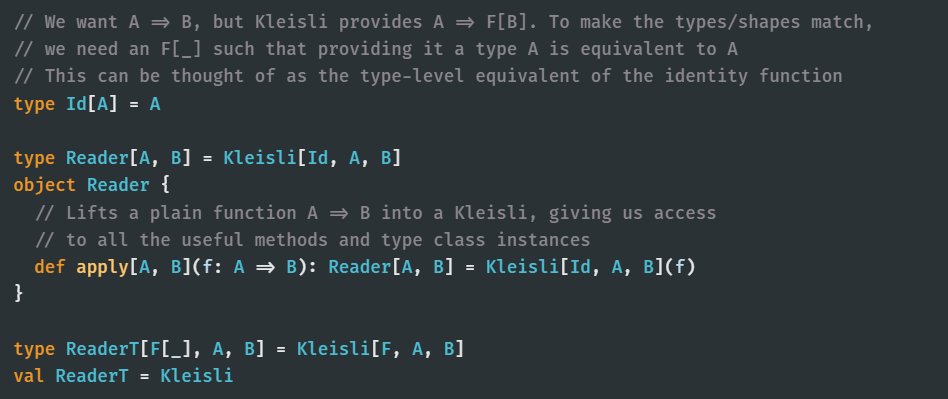
**Monad Transformers:**

Many data types have a monad transformer equivalent that allows us to compose the Monad instance of the data type with any other Monad instance. For instance, OptionT[F[\_], A] allows us to compose the monadic properties of Option with any other F[\_], such as a List. This allows us to work with nested contexts/effects in a nice way (for example, in for-comprehensions).

Kleisli can be viewed as the monad transformer for functions, as Kleisli[F, A, B] is just a function A => F[B]. Kleisli allows us to take the effects of functions and have them play nice with the effects of any other F[\_].

The effect of a function:  
- Takes an input (the input is a read-only copy, as assume function is pure)  
- Produces an output, based on this input

For this reason, the type class instances for functions often refer to the function as a Reader (Reader Monad).



**Configuration:**

Functional programming advocates the creation of programs and modules by composing smaller, simpler modules - write many small functions, and compose them to build larger ones.

*An example of this is shown in the code…*

Following on the code:

If wanted a module that doesn't need any config validation (e.g. logging events), then have this module instantiated from a config directly, without an Option.  
For example, have Kleisli[Id, LogConfig, Log] (alternatively, Reader[LogConfig, Log]). However, this won't integrate well with other Kleislis since those use Option instead of Id.

Kleisli also has a *lift* method that takes a type parameter G[\_] such that G has an Applicative instance and lifts a Kleisli value such that its output type is G[F[B]] – lifts/wraps the Kleisli in the new G[\_] type.  
Therefore, can lift a Reader[A, B] into a Kleisli[G, A, B].  
Note that lifting a Reader[A, B] into some G[\_] is equivalent to having a Kleisli[G, A, B] since Reader[A, B] is just a type alias for Kleisli[Id, A, B], and type Id[A] = A so G[Id[A]] is equivalent to G[A].