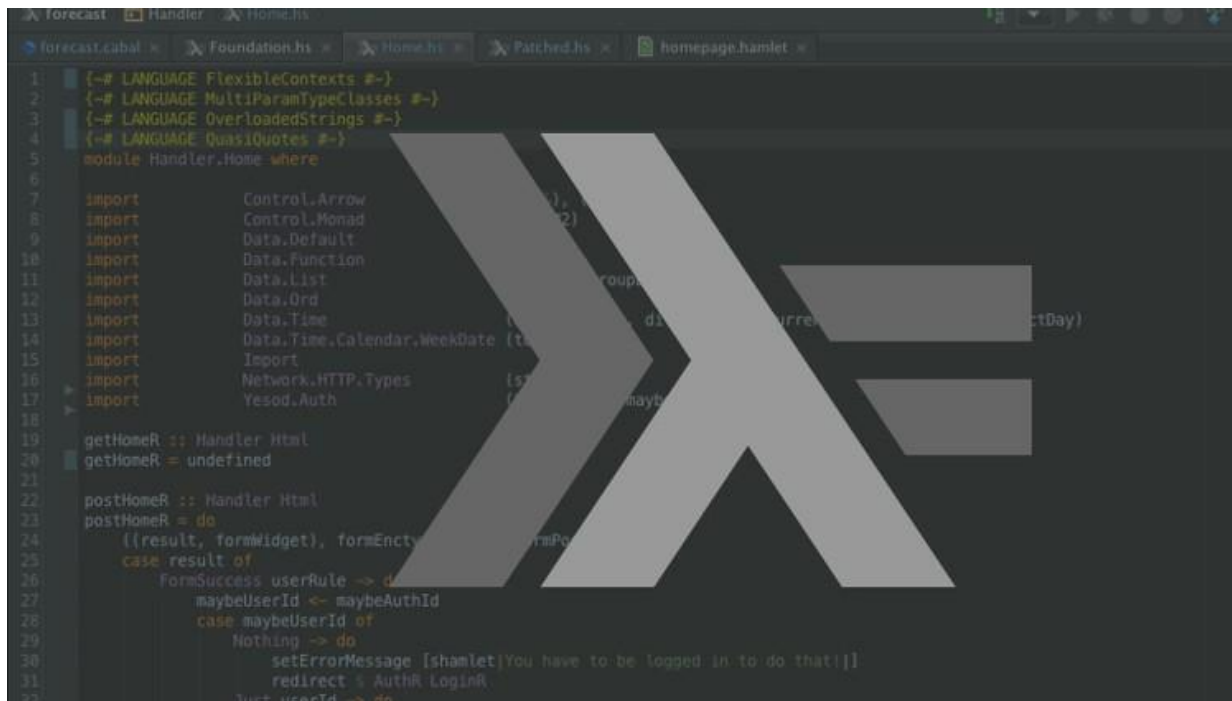


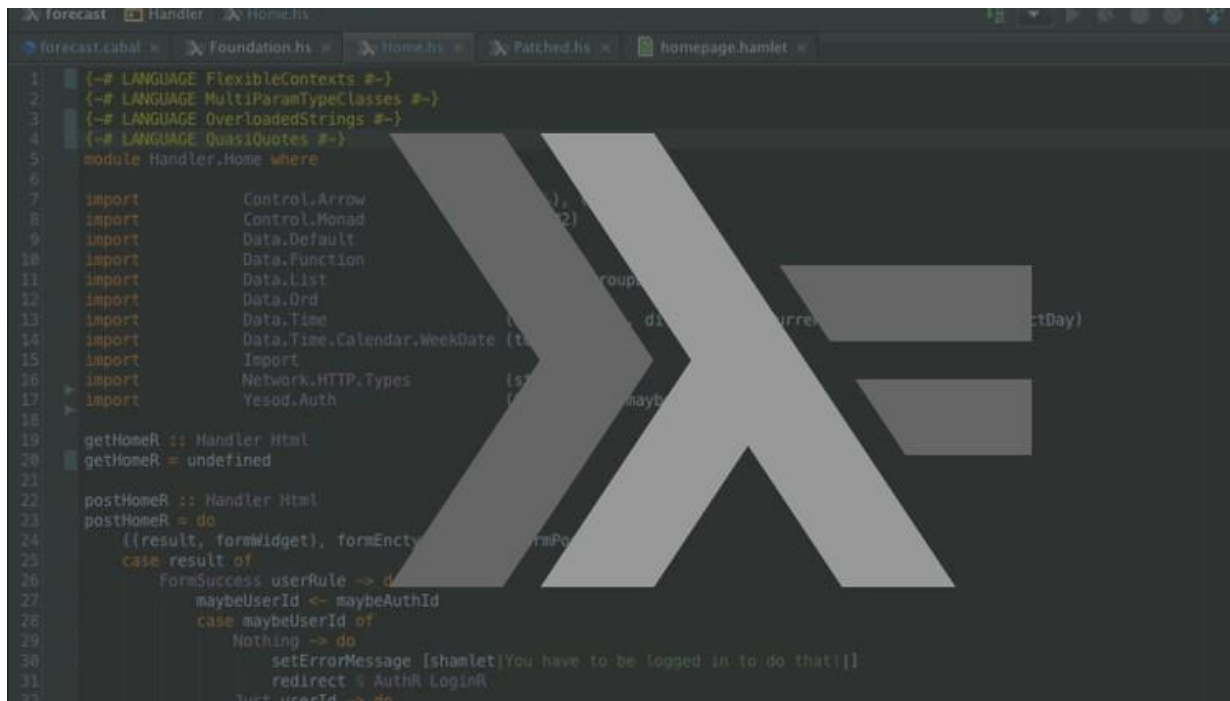
Monads



The image shows a screenshot of a Haskell code editor with a dark theme. The editor has several tabs at the top: 'forecast.cabal', 'Foundation.hs', 'Home.hs', 'Patched.hs', and 'homepage.hamlet'. The 'Home.hs' tab is active, displaying Haskell code. A large, semi-transparent watermark with the letters 'X' and 'E' is overlaid on the code. The code defines a module 'Handler.Home' and imports various libraries including 'Control.Monad', 'Data.Default', 'Data.Function', 'Data.List', 'Data.Ord', 'Data.Time', 'Data.Time.Calendar.WeekDate', 'Network.HTTP.Types', and 'Yesod.Auth'. It defines two handlers, 'getHomeR' and 'postHomeR', using monadic constructs like 'do' and 'case' expressions. The 'postHomeR' handler uses 'formEncrypted' and 'maybeAuthId' to handle user authentication.

```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Monad
8 import Control.Monad
9 import Data.Default
10 import Data.Function
11 import Data.List
12 import Data.Ord
13 import Data.Time
14 import Data.Time.Calendar.WeekDate
15 import Network.HTTP.Types
16 import Yesod.Auth
17
18 getHomeR :: Handler Html
19 getHomeR = undefined
20
21 postHomeR :: Handler Html
22 postHomeR = do
23   ((result, formWidget), formEncrypted) <- formEncrypted
24   case result of
25     FormSuccess userRule -> do
26       maybeUserId <- maybeAuthId
27       case maybeUserId of
28         Nothing -> do
29           setErrorMessage [shamlet|You have to be logged in to do that!|]
30           redirect % AuthR.LoginR
31         Just userId -> do
```

Bind Operator (>>=)



```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Arrow ((>>=))
8 import Control.Monad
9 import Data.Default
10 import Data.Function
11 import Data.List
12 import Data.Ord
13 import Data.Time
14 import Data.Time.Calendar.WeekDate (toWeekDate)
15 import Import
16 import Network.HTTP.Types (isStatus200)
17 import Yesod.Auth
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEnctype) <- runFormPost
25   case result of
26     FormSuccess userRule -> do
27       maybeUserId <- maybeAuthId
28       case maybeUserId of
29         Nothing -> do
30           setErrorMessage [shamlet|You have to be logged in to do that!|]
31           redirect % AuthR.LoginR
32         Just userId -> do
```

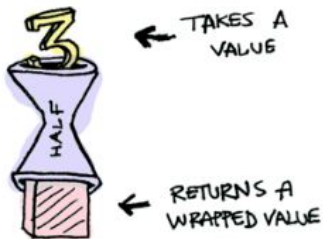
Monads



Consider that *half* is a function that only makes sense on even numbers:

```
half :: Int -> Maybe Int  
  
half x  
| even x    = Just (div x 2)  
| otherwise = Nothing
```

We can see the function like this: Given a value, return a packed value.



But then we can't stuff packed values into it!





We need a function that unpacks, applies `half` and leaves encapsulated.

This function is called `>=>` (pronounced *bind*)

```
λ> Just 40 >=> half      ➡ Just 20
λ> Just 31 >=> half      ➡ Nothing
λ> Nothing >=> half      ➡ Nothing

λ> Just 20 >=> half >=> half      ➡ Just 5
λ> Just 20 >=> half >=> half >=> half ➡ Nothing
```

The operator `>=>` is an operation of the class `Monad`:

```
class Applicative m => Monad m where
  (>=>) :: m a -> (a -> m b) -> m b
```

The type `Maybe` is instance of `Monad`:

```
instance Monad Maybe where
  Nothing >=> f    = Nothing
  Just x  >=> f    = f x
```

Instructor Youtube Channel: Lucas Science

