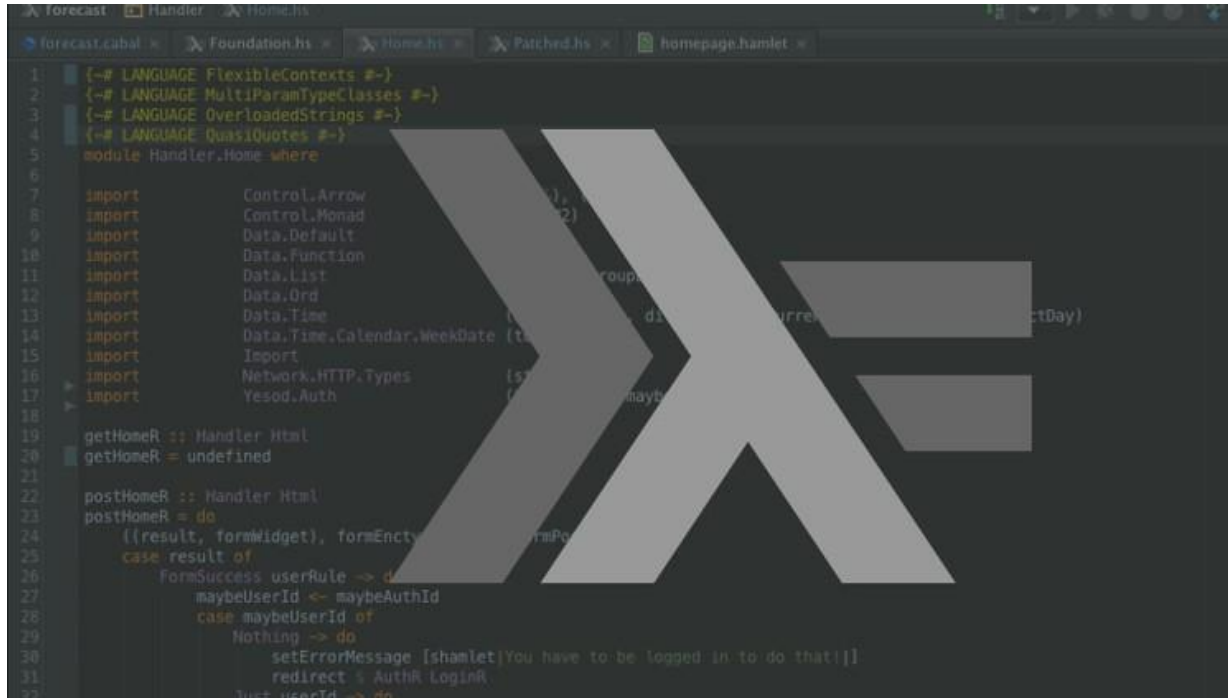


State Monad



```
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
15 import Import
16 import Network.HTTP.Types
17 import Yesod.Auth
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEnctype) <- runPost
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
```

State Monad



The **State Monad** in Haskell is a powerful tool for managing and manipulating state within a functional programming paradigm.

It allows us to write code that appears to be purely functional while still encapsulating state changes in a clean and predictable manner.

Understanding the State Monad is essential for functional programming in Haskell and can greatly enhance our ability to write elegant and maintainable code.

Concept of State



State represents the current condition or context of a program, and in purely functional programming, managing state can be challenging.

Concept of State



State represents the current condition or context of a program, and in purely functional programming, managing state can be challenging.

Example: Managing State Imperatively

Consider a simple counter that needs to be incremented or decremented. In imperative programming, we typically use mutable variables to manage state. For example, in Python:

```
counter = 0  
counter += 1
```



This mutable approach is not purely functional and can lead to various issues

Immutability and Referential Transparency



Now we will delve into two fundamental concepts in functional programming: **immutability** and **referential transparency**. Understanding these principles is essential for appreciating the challenges of managing state in a purely functional context.

Immutability



Immutability is a core principle of functional programming. It means that once data is created, it cannot be changed. In functional languages like Haskell, data structures are immutable by default. This immutability ensures that data remains constant throughout its lifetime, making it easier to reason about code and preventing unintended side effects.

In our example, when we attempted to increment a counter in Haskell without the State Monad, we violated immutability principles because we were trying to modify the counter's value directly.

Referential Transparency



Referential transparency is another key concept in functional programming. It means that a function's output depends solely on its input, and calling a function with the same inputs will always produce the same result. This property allows for predictable and reliable code because functions don't have hidden side effects.

State Monad



Let's define the **State Monad** and its essential components:

- ⦿ **State** type: Represents a stateful computation.
- ⦿ **State** constructor: Wraps a function that takes an initial state and produces a result and a new state.
- ⦿ **runState** function: Executes the stateful computation.

State Monad



```
newtype State s a = State { runState :: s -> (a, s) }
```

With the State Monad, we can manage state in a purely functional manner while maintaining the principles of **immutability** and **referential transparency**.