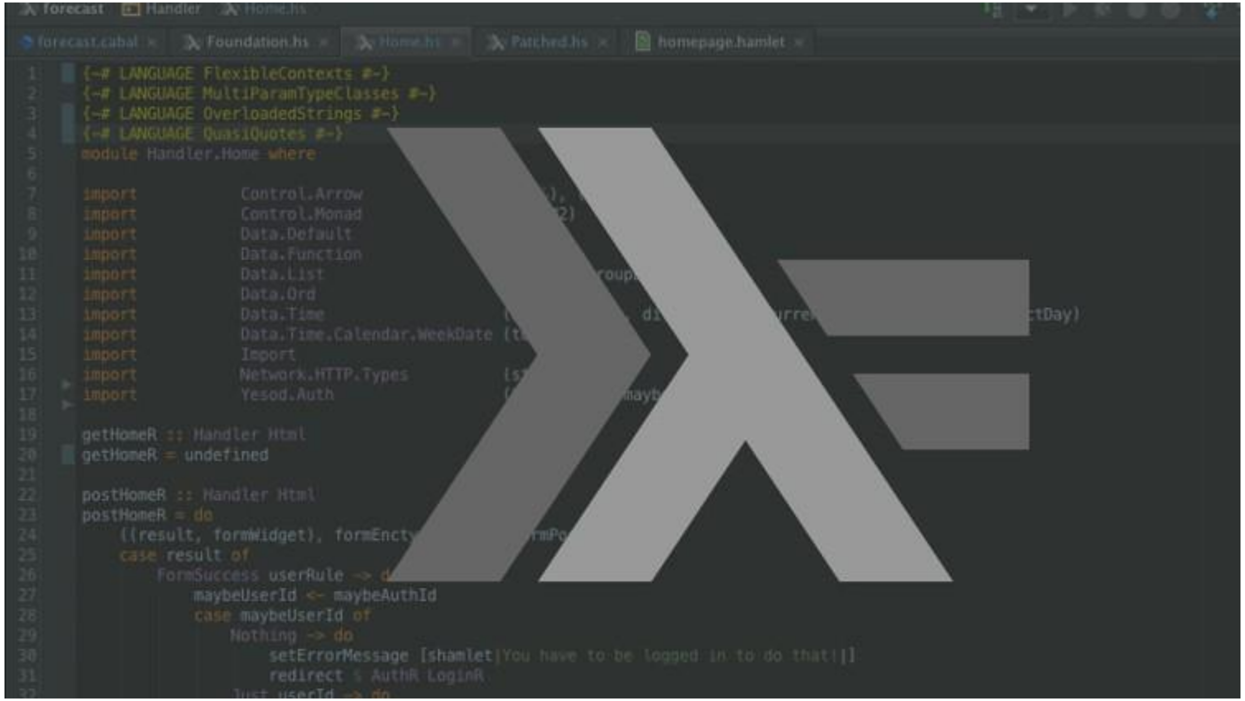
Anonymous Functions



```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

import           Control.Arrow                       ), (
import           Control.Monad                       2)
import           Data.Default
import           Data.Function
import           Data.List                           roup
import           Data.Ord
import           Data.Time                  (     , di      urre               tDay)
import           Data.Time.Calendar.WeekDate (t
import           Import
import           Network.HTTP.Types         (s
import           Yesod.Auth                 (            mayb

getHomeR :: Handler Html
getHomeR = undefined

postHomeR :: Handler Html
postHomeR = do
    ((result, formWidget), formEncty           rmPo
    case result of
        FormSuccess userRule -> d
            maybeUserId <- maybeAuthId
            case maybeUserId of
                Nothing -> do
                    setErrorMessage [shamlet|You have to be logged in to do that!|]
                    redirect $ AuthR LoginR
                Just userId -> do
```

Lucas Bazilio - Udemy

Anonymous Functions

Anonymous Functions (λ functions) are expressions that represent a function without name.

```
\x -> x + 5             -- defines anonymous function that, given a x, returns x + 5
                        -- si proveu d'escriure-la, Haskell s'enfada perquè no ho sap fer

(\x -> x + 5) 3         applies the anonymous function over 3
☞ 8
```

Function with name:

```
double x = 2 * x                 -- equals to double = \x -> 2 * x

λ> double 3                      ☞ 6

λ> map double [1, 2, 3]          ☞ [2, 4, 6]
```

Lucas Bazilio - Udemy

Anonymous Functions

Anonymous Function:

```
λ> map (\x -> 2 * x) [1, 2, 3]      ☞ [2, 4, 6]
```

Anonymous functions are usually used when they are short and only used once. They are also useful for performing program transformations.

Anonymous Functions

Multiple parameters:

```
\x y -> x + y
```

equals to

```
\x -> \y -> x + y
```

which means

```
\x -> (\y -> x + y)
```

Instructor Youtube Channel: Lucas Science

Lucas Science

SUBSCRIBE