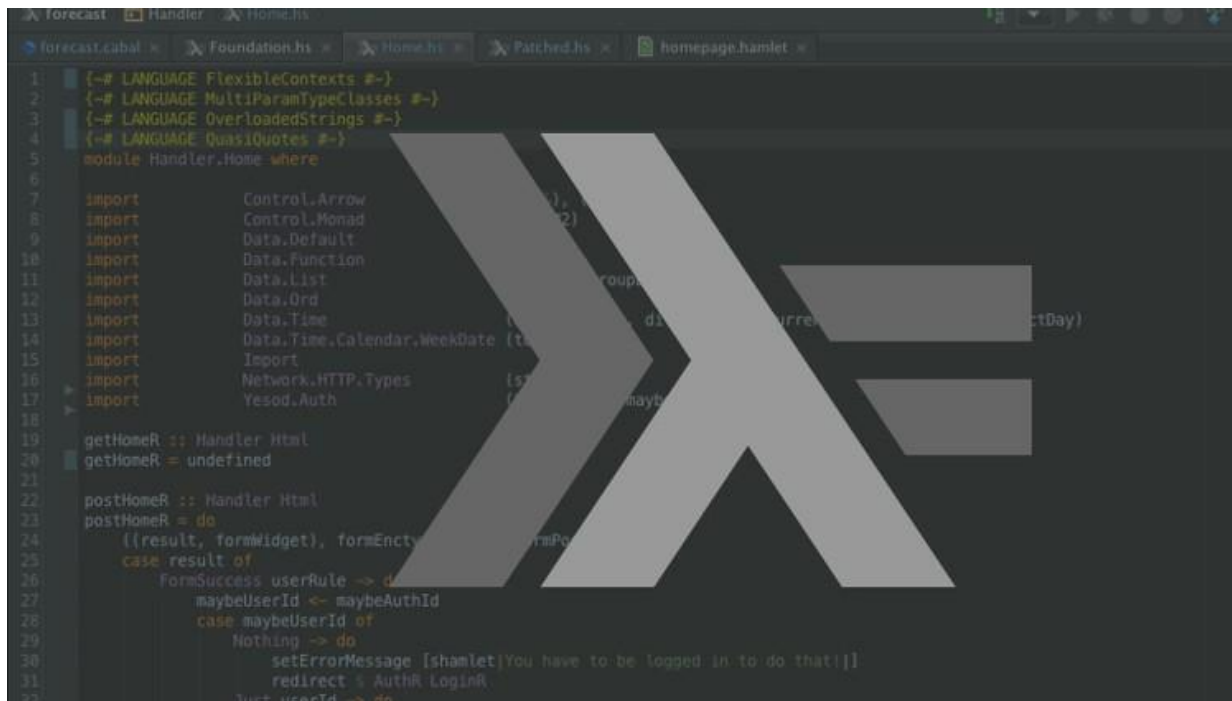


# Higher Order Functions



```
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), formEnctype) <- runFormPost
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
```

# Higher Order Functions

A **Higher Order Function** (HOF) is a function that receives or returns functions

Key point: Functions are first-class objects.

**Example in C++:**

```
bool compare(int x, int y) {  
    return x > y;  
}  
  
int main() {  
    vector<int> v = { ... };  
    sort(v.begin(), v.end(), compare);           // sort is a higher order function  
}
```

# Higher Order Functions

**Example:** The predefined function `map` applies a function to each element of a list.

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [] = []
```

```
map f (x:xs) = f x : map f xs
```

```
λ> map odd [1..5]
```

```
↵ [True, False, True, False, True]
```

# Higher Order Functions

**Example:** The predefined function `(.)` returns the composition of two functions:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

```
(f . g) x = f (g x)
```

```
λ> (reverse . sort) [5, 3, 5, 2]
```

```
↵ [5, 5, 3, 2]
```

# Higher Order Functions

**Example:** The `apli2` function applies a function to an element twice.

```
apli2 :: (a -> a) -> a -> a
```

```
apli2 f x = f (f x)
```

```
λ> apli2 sqrt 16.0  
↳ 2.0
```

Equivalently:

```
apli2 :: (a -> a) -> (a -> a)
```

```
apli2 f = f . f
```

```
λ> apli2 sqrt 16.0  
↳ 2.0
```

# Instructor Youtube Channel: Lucas Science

