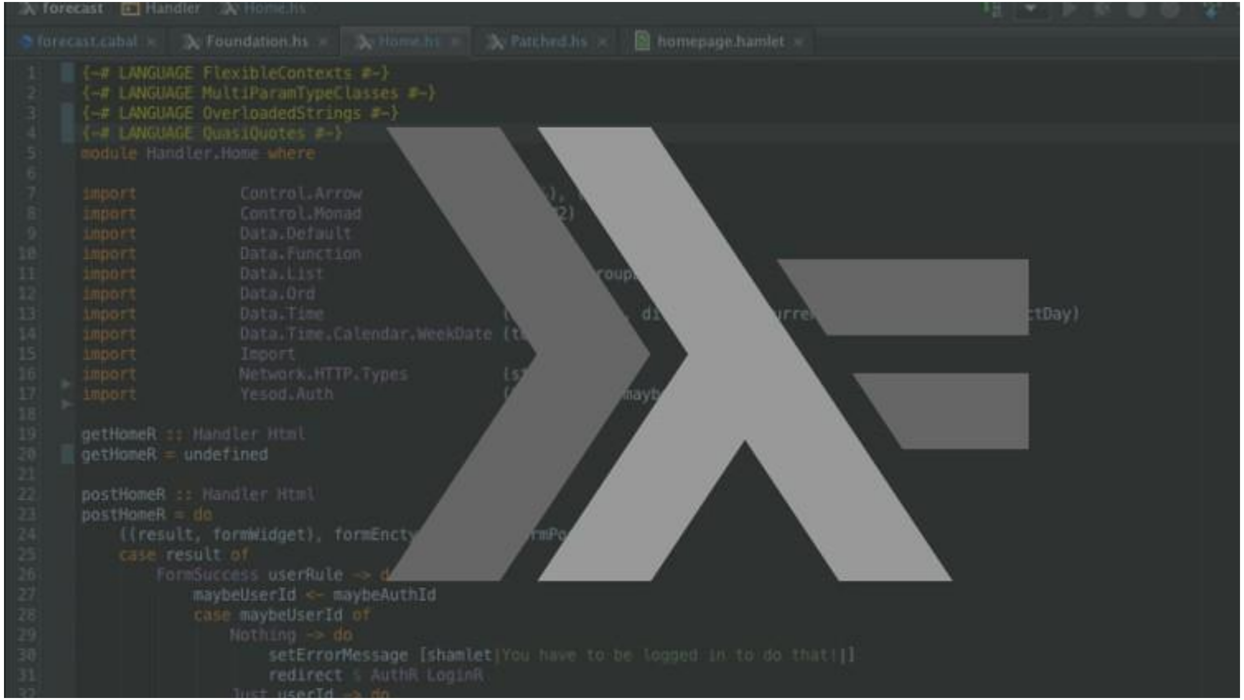
Final Exams



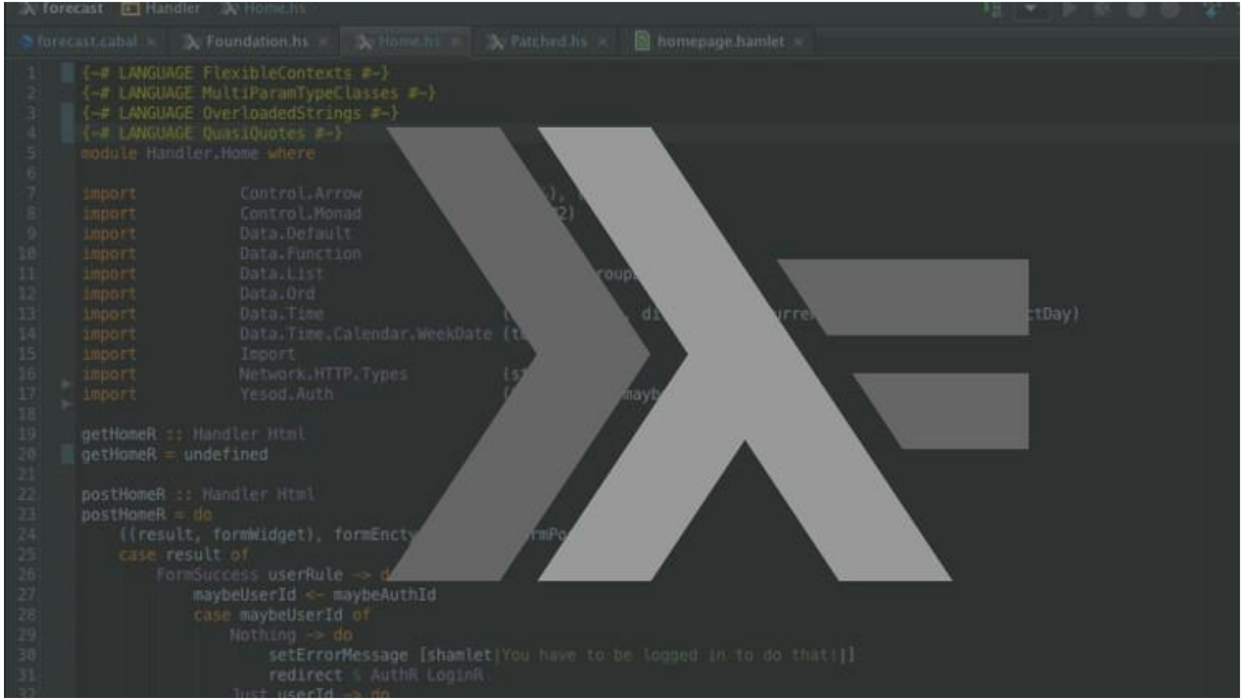
```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

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                    ctDay)
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
```

Exam 3



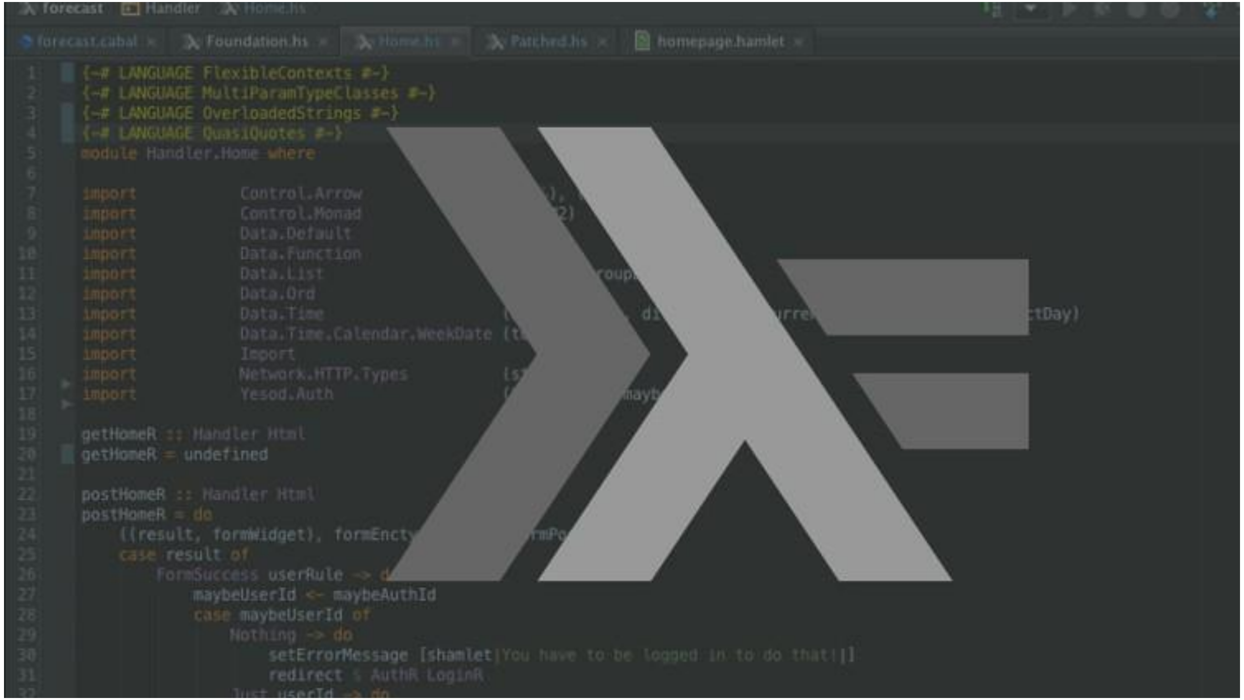
```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

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              ctDay)
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
```

Exam 3 - Problem 1



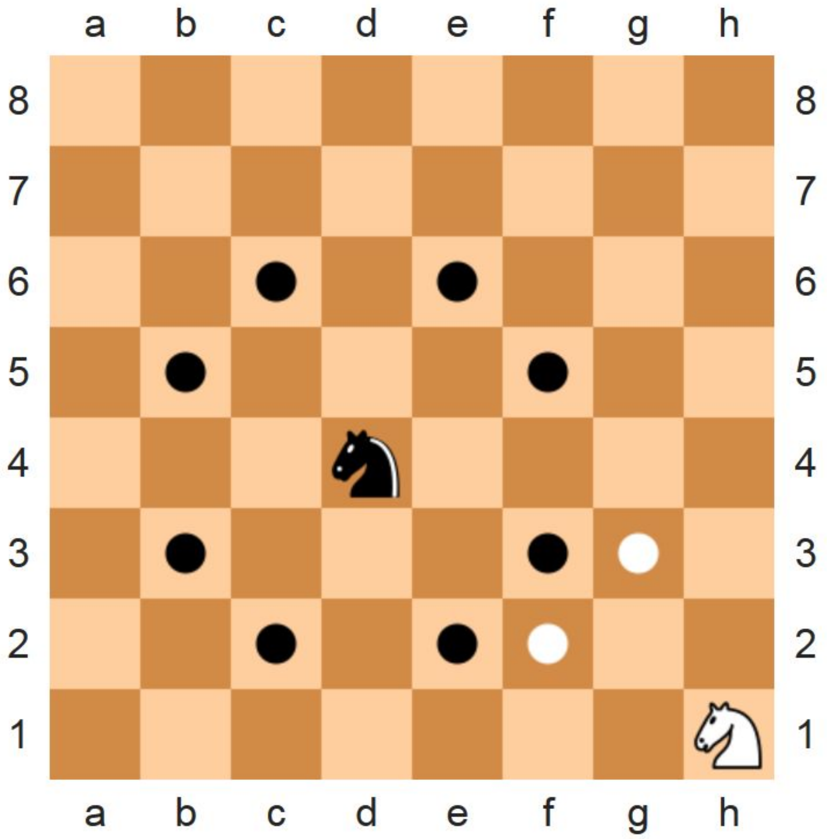
```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

import          Control.Arrow                  ), (
import          Control.Monad                  2)
import          Data.Default
import          Data.Function
import          Data.List                        roup
import          Data.Time                      (      , di      urre                    ctDay)
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
```

Exam 3 - Problem 1



Problem 1

Consider a knight on an empty 8×8 chess board. Its position can be given with a tuple indicating its row and column:

```
type Pos = (Int, Int) -- bottom left box is (1,1)
```

Remember that knights move in an "L":

1. Define a function *inside :: Pos -> Bool* that, given a position of a horse, returns if it is inside the board.

2. Define a function moves :: Pos -> [Pos] that, given a position of a horse within the board, returns the list of positions within the board where it can be found after a jump. The order of the list is not important: Test sets already sort it with luck. But you must write import Data.List(sort) at the beginning of your program.

Problem 1

Consider a knight on an empty 8×8 chess board. Its position can be given with a tuple indicating its row and column:

```
type Pos = (Int, Int) -- bottom left box is (1,1)
```

Remember that knights move in an "L":

3. Define a function *canGo3 :: Pos -> Pos -> Bool* that, given a start position p within the board and a final position q, tells whether a horse can go from p to q in (exactly) three jumps.

4. Now define a function *canGo3' :: Pos -> Pos -> Bool* that does the same as canGo3 but taking advantage of the fact that lists are Monad instances.

Problem 1

Public Test Case

Input	**Output**
inside (4, 5)	True
inside (0, 1)	False
inside (4, 9)	False
sort $ moves (4, 5)	[(2,4),(2,6),(3,3),(3,7),(5,3),(5,7),(6,4),(6,6)]
sort $ moves (1, 1)	[(2,3),(3,2)]
canGo3 (1, 1) (4, 5)	True
canGo3 (1, 1) (4, 6)	False
canGo3' (1, 1) (4, 5)	True
canGo3' (1, 1) (4, 6)	False