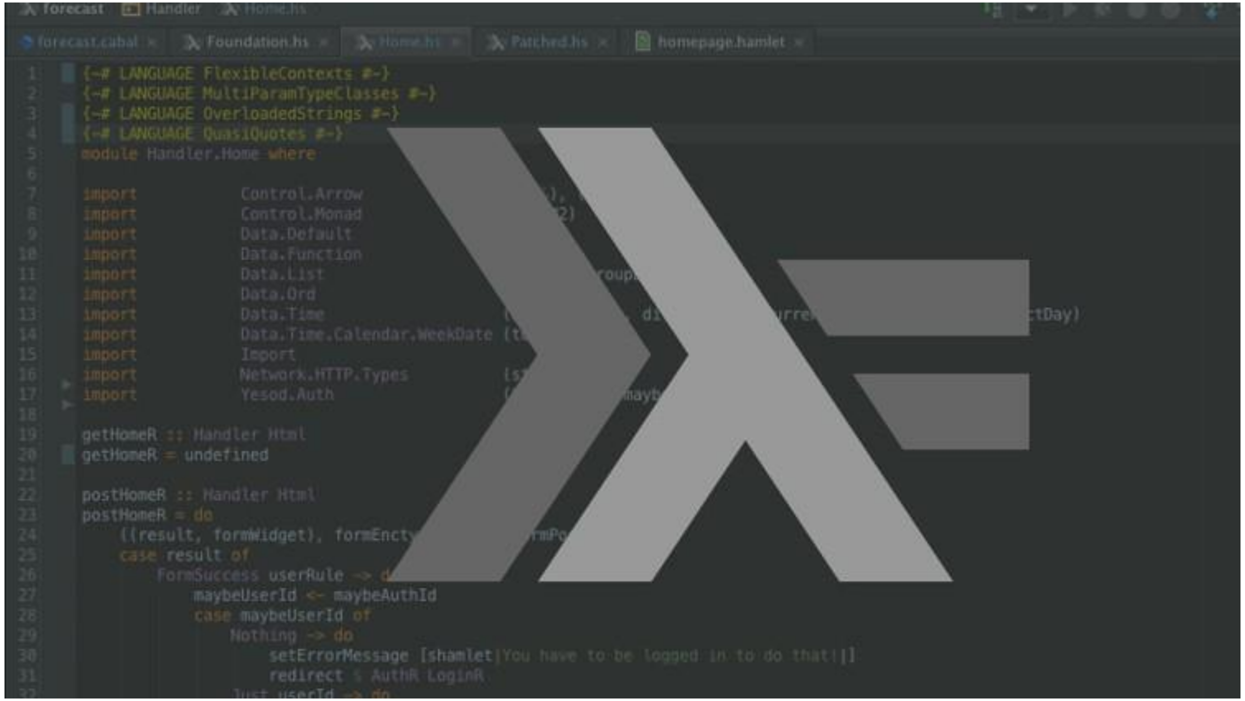
Introduction to Functions



```
{-# 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.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), formEnct            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

Functions in Haskell

- Functions in Haskell are *pures*: they only return results calculated relative to their parameters.
- Functions do not have *side effects*.
    - they do not modify the parameters
    - they do not modify the memory
    - they do not modify the input/output
- A function always returns the same result applied to the same parameters.

Definition of Functions

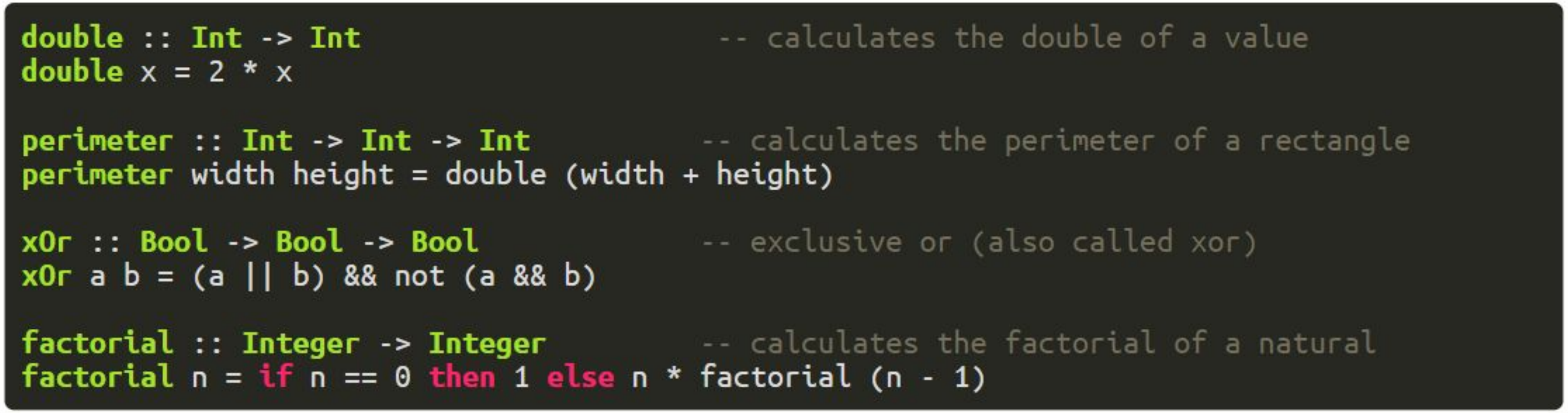
Function identifiers start with a lowercase.

To introduce a function:

1. First, its type declaration (header) is given.
2. Then its definition is given, using formal parameters.

Definition of Functions

Examples:



```haskell
double :: Int -> Int                     -- calculates the double of a value
double x = 2 * x

perimeter :: Int -> Int -> Int          -- calculates the perimeter of a rectangle
perimeter width height = double (width + height)

xOr :: Bool -> Bool -> Bool             -- exclusive or (also called xor)
xOr a b = (a || b) && not (a && b)

factorial :: Integer -> Integer         -- calculates the factorial of a natural
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

Lucas Bazilio - Udemy