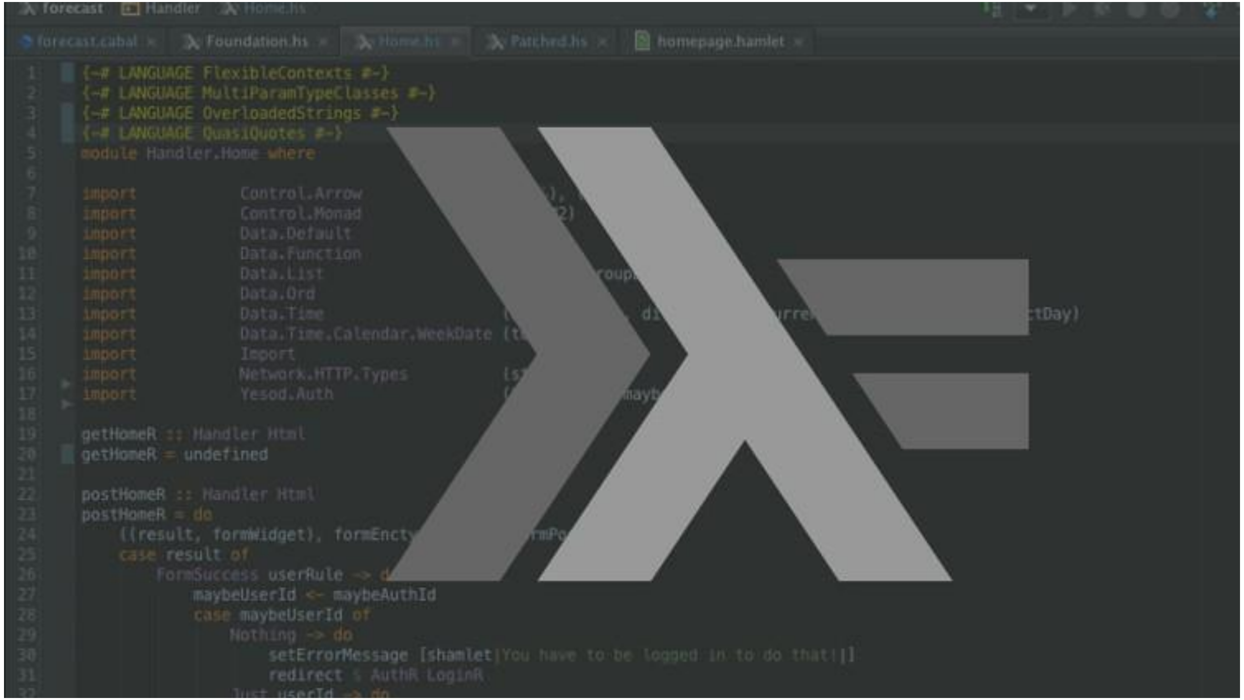
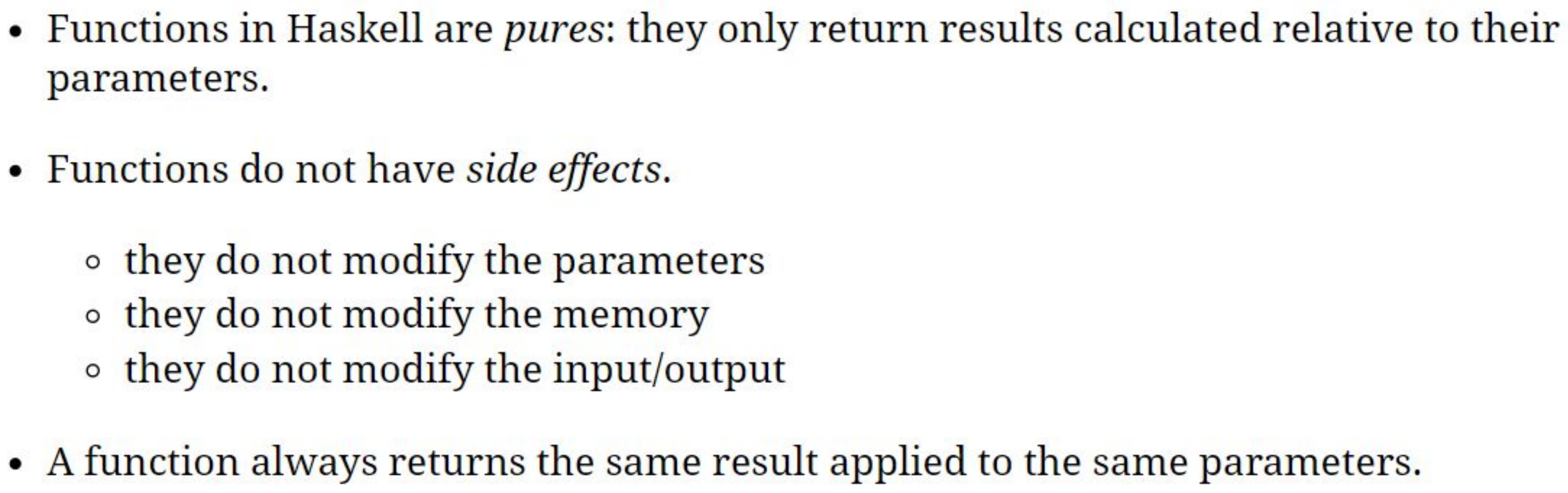
Introduction to Functions



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

import          Control.Arrow
import          Control.Monad
import          Data.Default
import          Data.Function
import          Data.List
import          Data.Ord
import          Data.Time
import          Data.Time.Calendar.WeekDate (t
import          Import
import          Network.HTTP.Types          (s
import          Yesod.Auth                  (

getHomeR :: Handler Html
getHomeR = undefined

postHomeR :: Handler Html
postHomeR = do
    ((result, formWidget), formEncty
    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

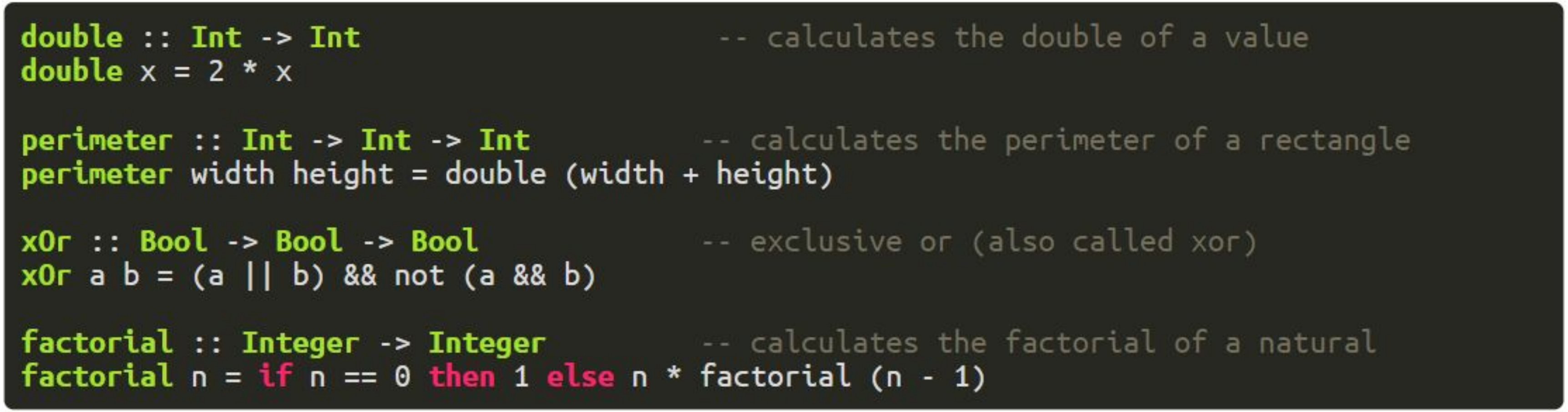
Function identifiers start with a lowercase.

To introduce a function:

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

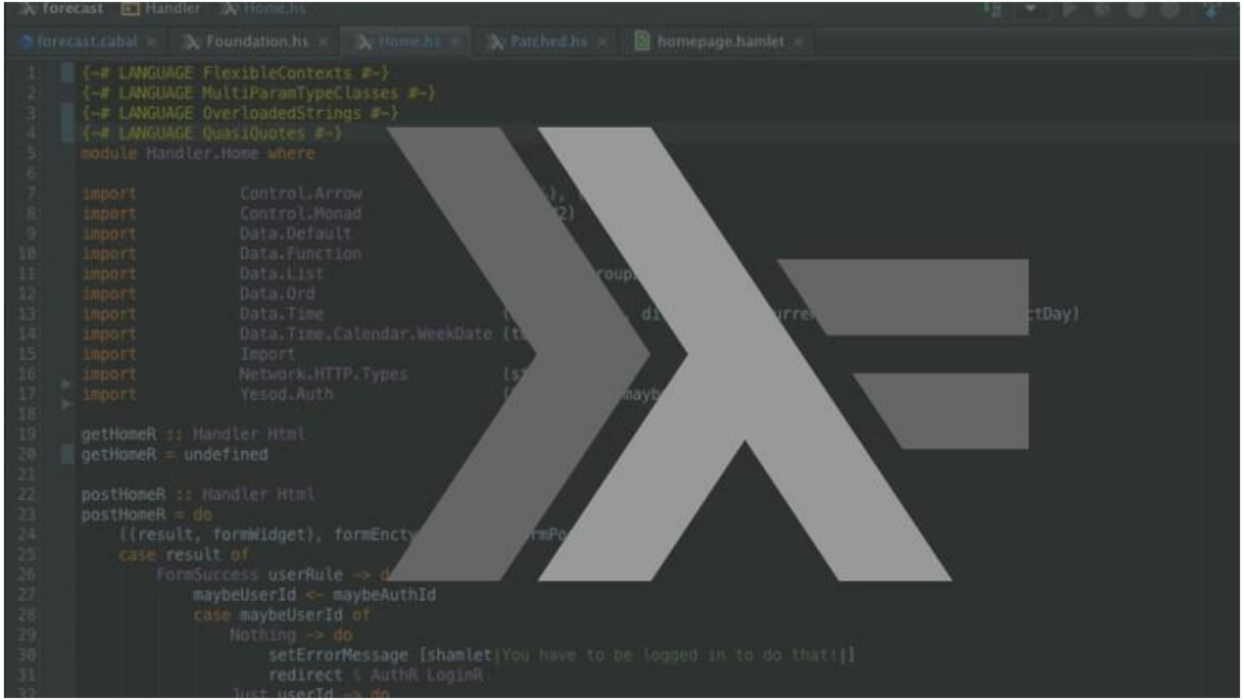
Definition of Functions

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)
```

Definition with Patterns

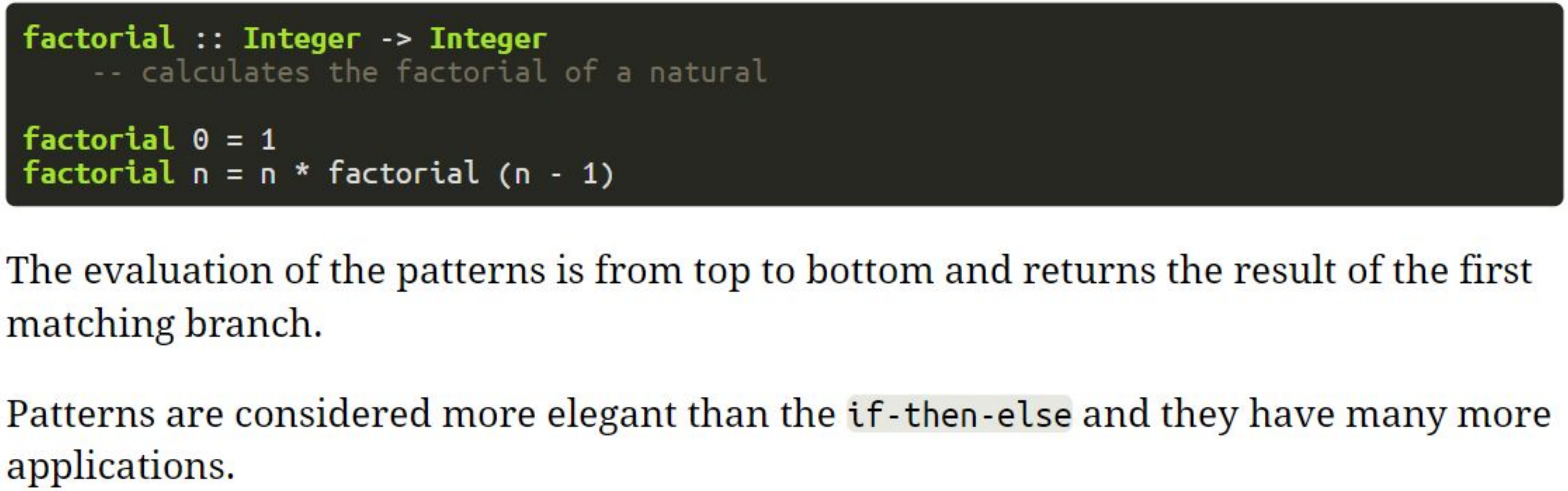


```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

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
```

Lucas Bazilio - Udemy

Definition with Patterns

Functions can be defined with patterns:

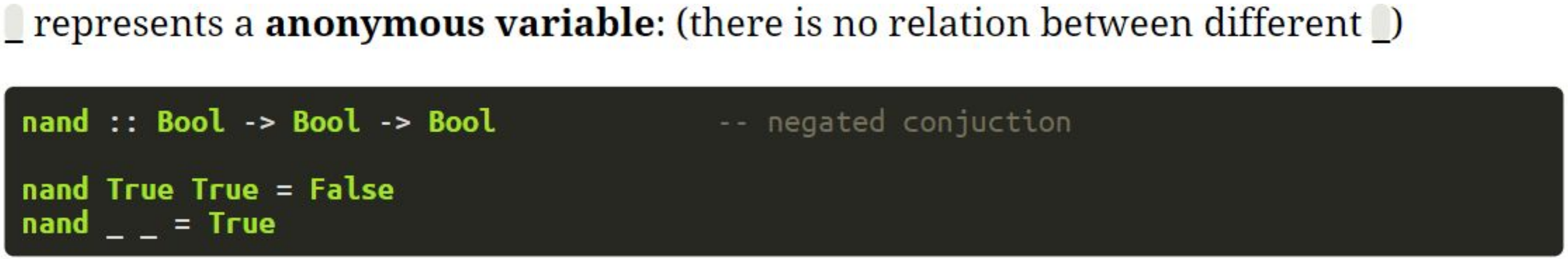


```
factorial :: Integer -> Integer
    -- calculates the factorial of a natural

factorial 0 = 1
factorial n = n * factorial (n - 1)
```

The evaluation of the patterns is from top to bottom and returns the result of the first matching branch.

Patterns are considered more elegant than the `if-then-else` and they have many more applications.

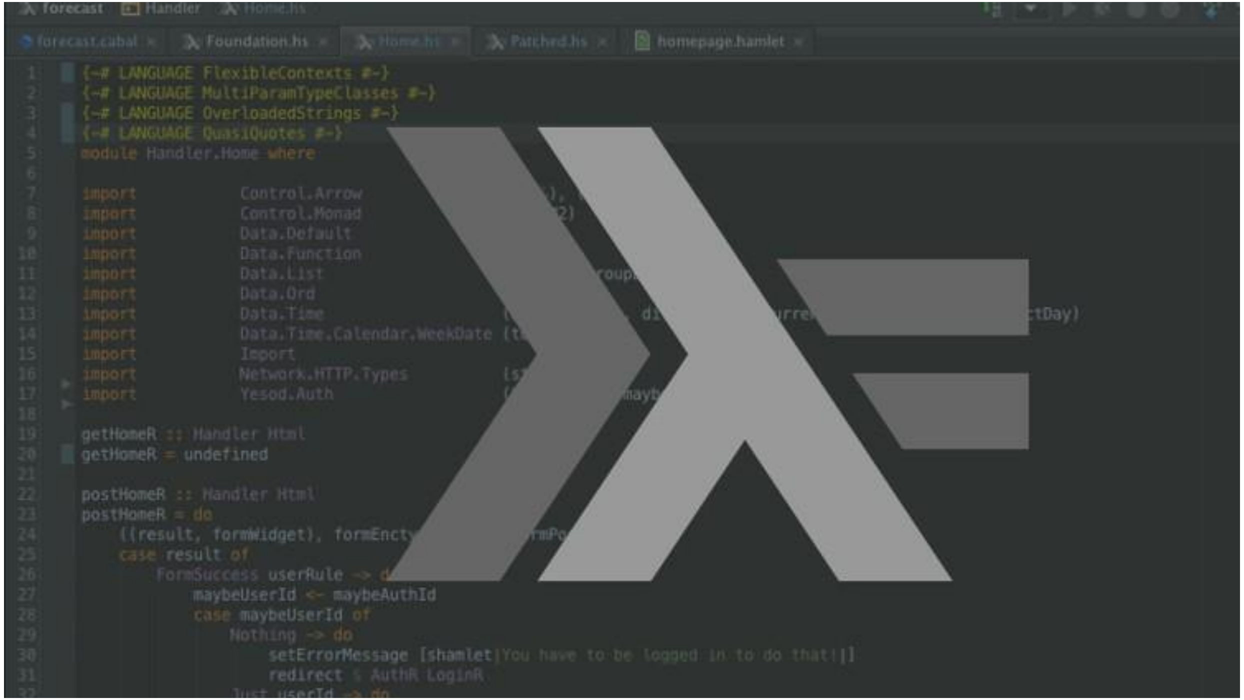
Definition with Patterns



_ represents a **anonymous variable**: (there is no relation between different _)

```
nand :: Bool -> Bool -> Bool                -- negated conjuction

nand True True = False
nand _ _ = True
```

Definition with Guards

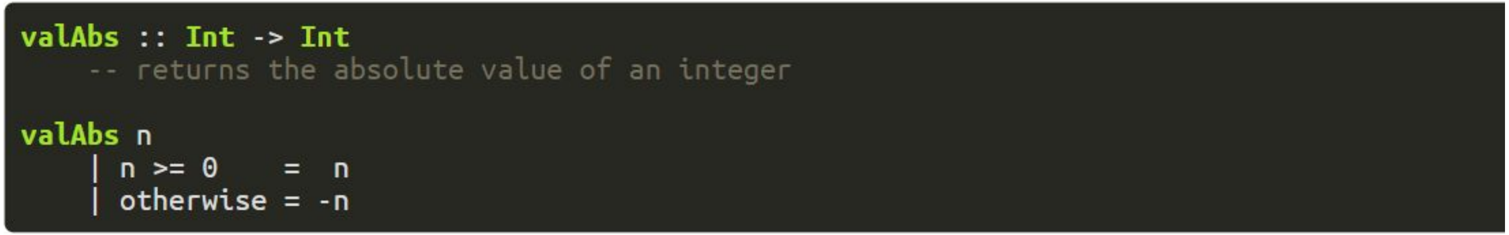


```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

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
```

Lucas Bazilio - Udemy

Definition with Guards

Functions can be defined with **guards**:



```haskell
valAbs :: Int -> Int
    -- returns the absolute value of an integer

valAbs n
    | n >= 0    =  n
    | otherwise = -n
```

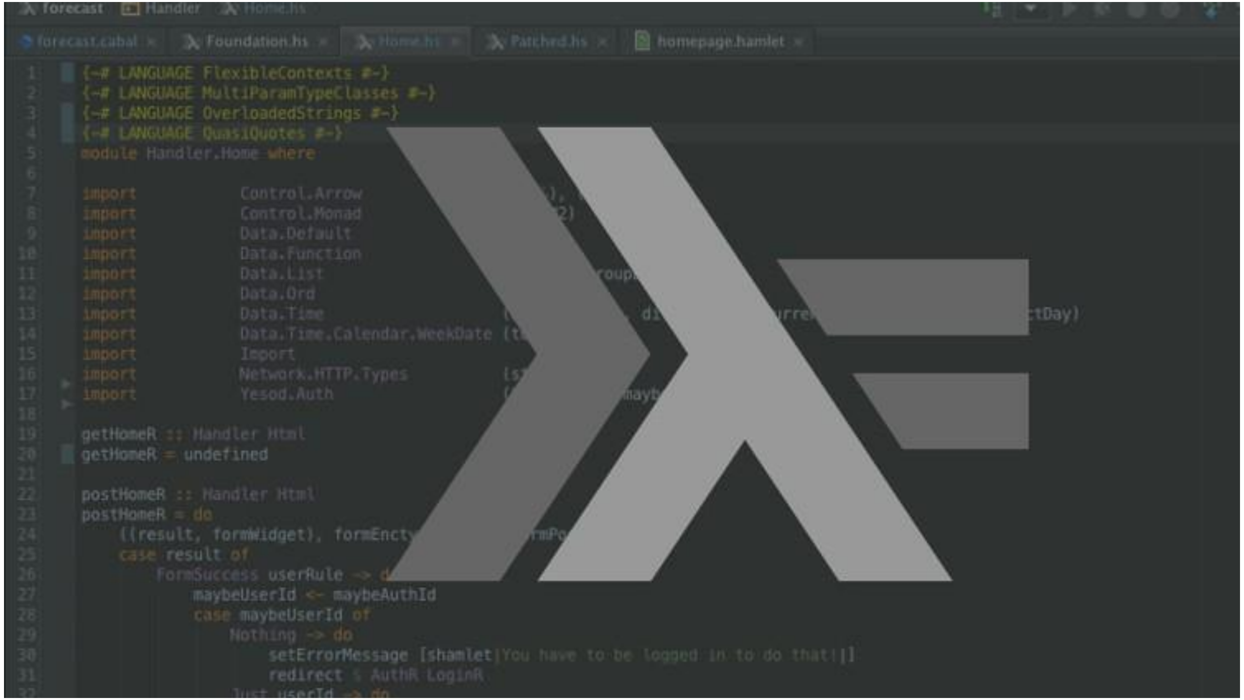
Guard evaluation is top-down and returns the result of the first true branch.

Pattern definitions can also have guards.

The `otherwise` is the same as `True`, but more readable.

⚠ Equality goes after every guard!

Local Definitions

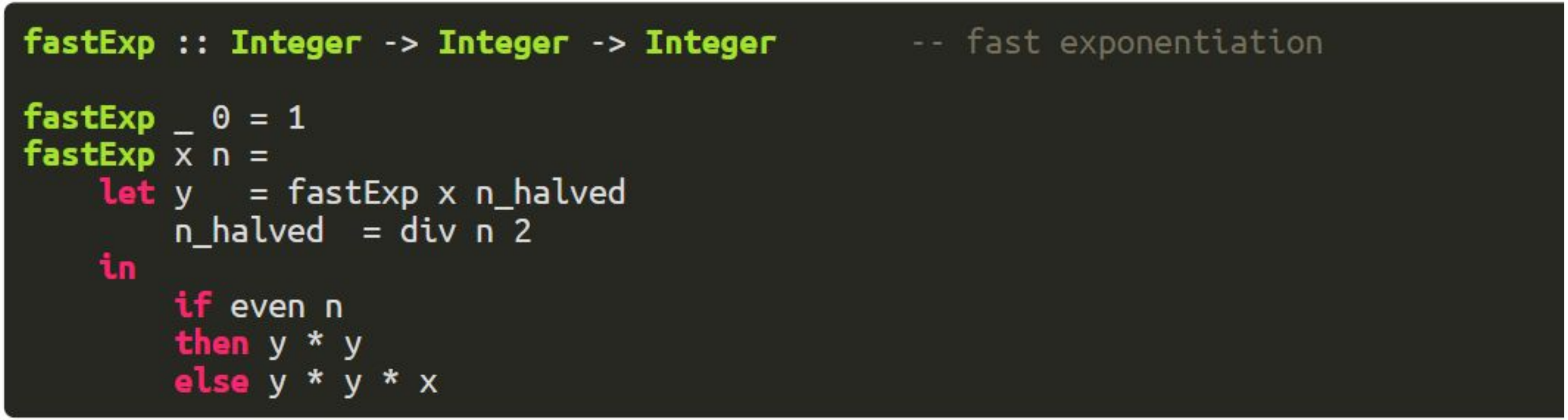


```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

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                  (                maybe

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

Local Definitions

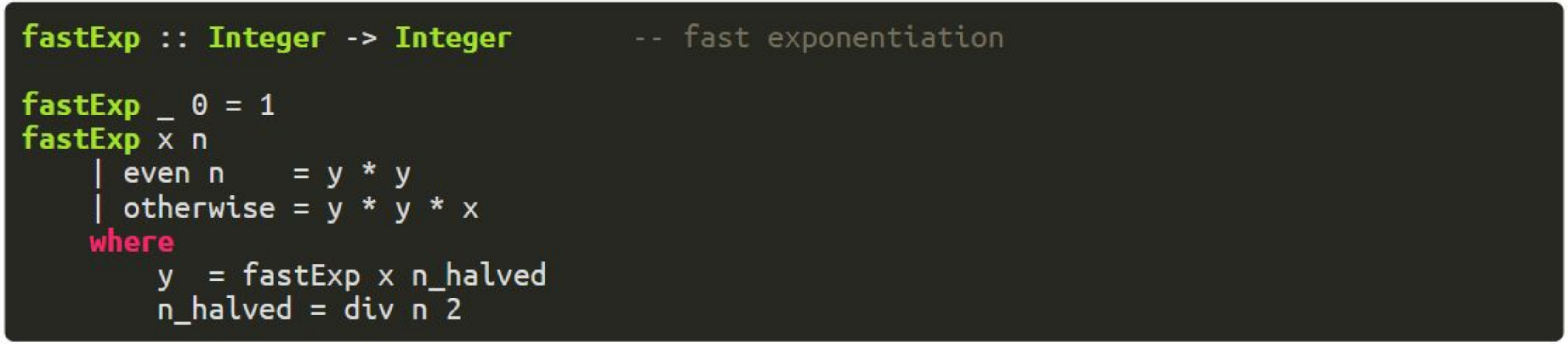
To define local names in an expression it is used the `let-in`:



```haskell
fastExp :: Integer -> Integer -> Integer       -- fast exponentiation

fastExp _ 0 = 1
fastExp x n =
    let y   = fastExp x n_halved
        n_halved  = div n 2
    in
        if even n
        then y * y
        else y * y * x
```

Local Definitions

The `where` allows names to be defined in more than one expression:



```haskell
fastExp :: Integer -> Integer        -- fast exponentiation

fastExp _ 0 = 1
fastExp x n
    | even n    = y * y
    | otherwise = y * y * x
    where
        y  = fastExp x n_halved
        n_halved = div n 2
```

The indentation of the `where` defines its scope.