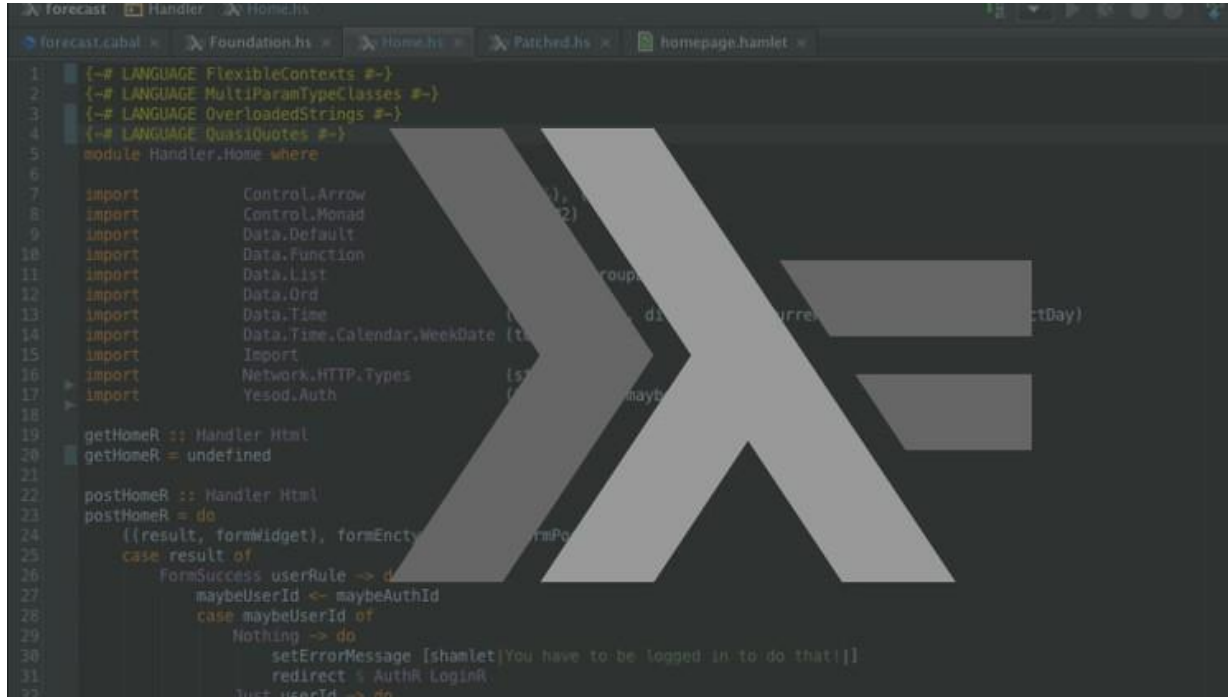


# Introduction to Functions



```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Arrow
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 Import
16 import Network.HTTP.Types
17 import Yesod.Auth
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEnctype) <- runFormPost
25   case result of
26     FormSuccess userRule -> do
27       maybeUserId <- maybeAuthId
28       case maybeUserId of
29         Nothing -> do
30           setErrorMessage [shamlet|You have to be logged in to do that!|]
31           redirect % AuthR.LoginR
32         Just userId -> do
```

# 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:

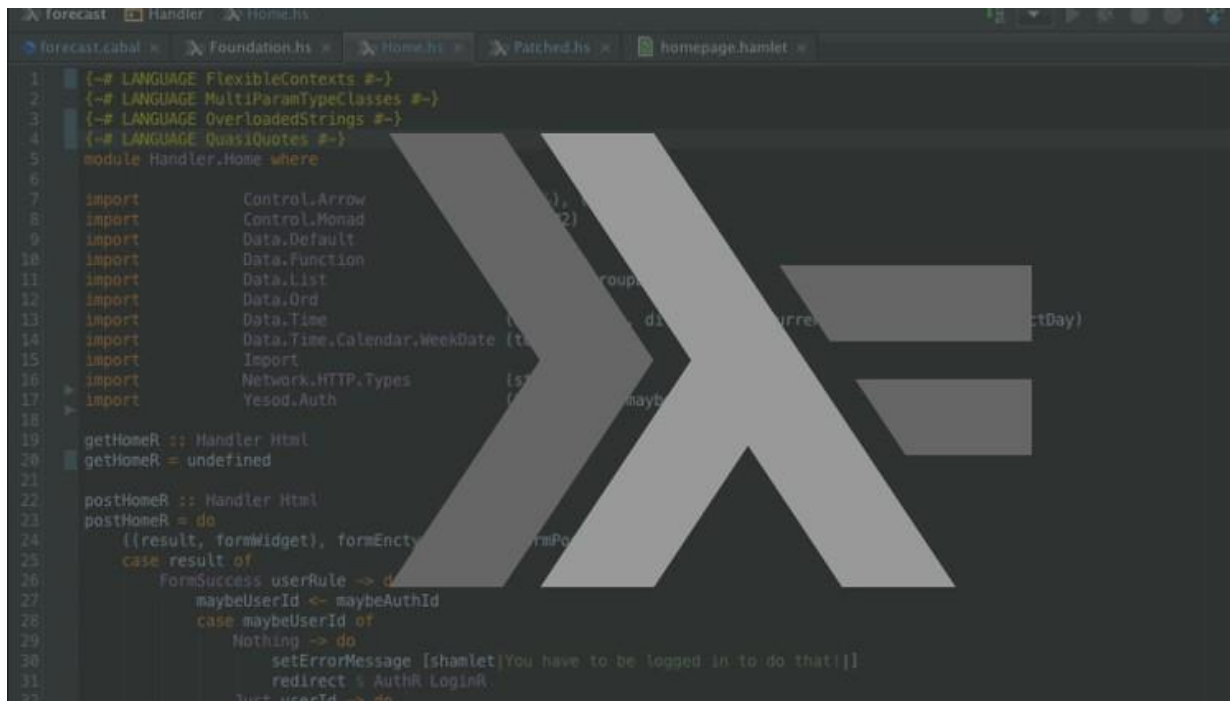
```
double :: Int -> Int           -- calculates the double of a value
double x = 2 * x

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



```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Arrow
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 Import
16 import Network.HTTP.Types
17 import Yesod.Auth
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEnctype) <- runFormPost
25   case result of
26     FormSuccess userRule -> do
27       maybeUserId <- maybeAuthId
28       case maybeUserId of
29         Nothing -> do
30           setErrorMessage [shamlet|You have to be logged in to do that!|]
31           redirect % AuthR.LoginR
32         Just userId -> do
```

# 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.

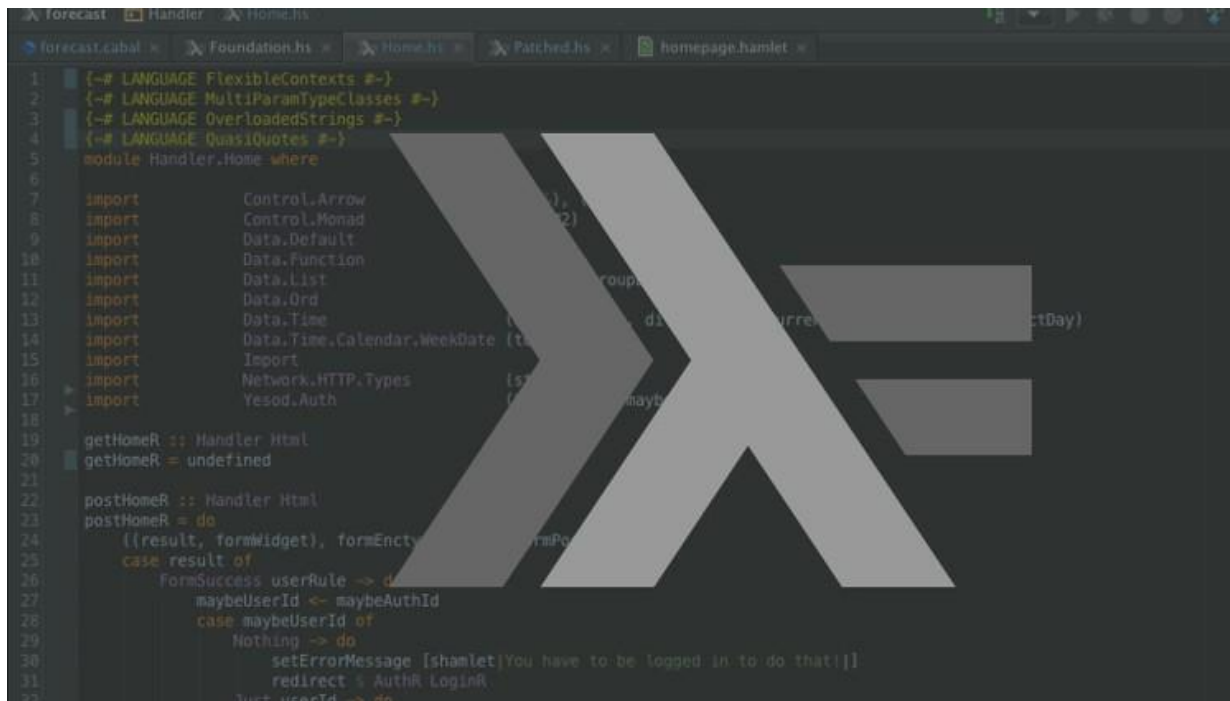
Patterns are considered more elegant than the `if-then-else` and they have many more applications.

# Definition with Patterns

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

```
nand :: Bool -> Bool -> Bool           -- negated conjunction
nand True True = False
nand _ _ = True
```

# Definition with Guards



```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Arrow
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 (toDayOfYear, dayOfWeek)
15 import Import
16 import Network.HTTP.Types
17 import Yesod.Auth
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEnctype) <- runFormPost
25   case result of
26     FormSuccess userRule -> do
27       maybeUserId <- maybeAuthId
28       case maybeUserId of
29         Nothing -> do
30           setErrorMessage [shamlet|You have to be logged in to do that!|]
31           redirect % AuthR.LoginR
32         Just userId -> do
```



# Definition with Guards

Functions can be defined with **guards**:

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

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

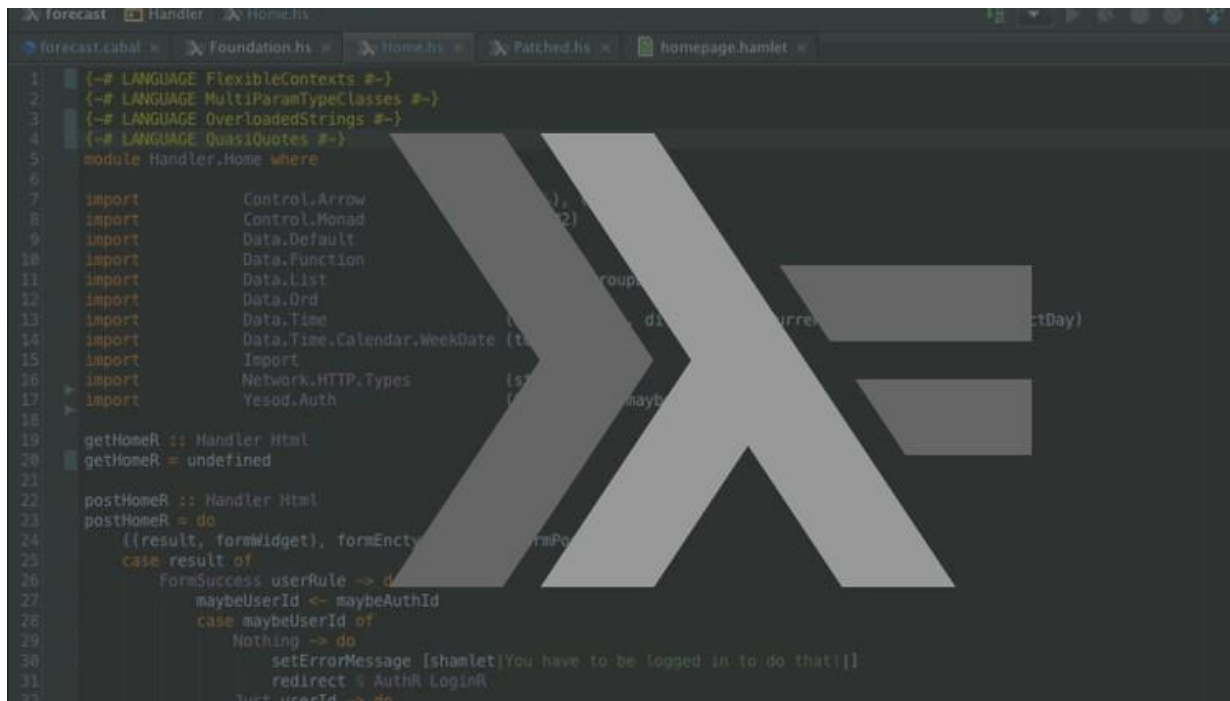
Guard evaluation is top-down and returns the result of the first true branch.

Pattern definitions can also have guards.

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

△ Equality goes after every guard!

# Local Definitions



The image shows a screenshot of a Haskell code editor with a dark theme. The editor has several tabs at the top: 'forecast.cabal', 'Foundation.hs', 'Home.hs', 'Patched.hs', and 'homepage.hamlet'. The 'Home.hs' tab is active, showing a Haskell module named 'Handler.Home'. The code includes several language pragmas for type classes, a list of imports from various modules like 'Control.Arrow', 'Data.Default', 'Data.Function', 'Data.List', 'Data.Map', 'Data.Maybe', 'Data.Time', 'Data.Time.Calendar', 'Network.HTTP.Types', and 'Yesod.Auth'. It then defines two functions: 'getHomeR' and 'postHomeR'. 'getHomeR' is currently set to 'undefined'. 'postHomeR' is a more complex function that uses 'do' notation and 'case' expressions to handle user authentication and redirection. A large, semi-transparent 'X' watermark is overlaid on the right side of the code.

```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE MultiParamTypeClasses #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE QuasiQuotes #-}
5 module Handler.Home where
6
7 import Control.Arrow ((>>)), Control.Monad (liftM2)
8 import Data.Default (Default)
9 import Data.Function (on)
10 import Data.List (group)
11 import Data.Map (Map)
12 import Data.Ord (comparing)
13 import Data.Time (Day, Time, currentTime, getCurrentTime, toDayTime)
14 import Data.Time.Calendar.WeekDate (toWeekDate)
15 import Import
16 import Network.HTTP.Types (Status)
17 import Yesod.Auth (AuthM, maybeAuthId)
18
19 getHomeR :: Handler Html
20 getHomeR = undefined
21
22 postHomeR :: Handler Html
23 postHomeR = do
24   ((result, formWidget), formEncrypt, formPost) <- runFormPost
25   case result of
26     FormSuccess userRule -> do
27       maybeUserId <- maybeAuthId
28       case maybeUserId of
29         Nothing -> do
30           setErrorMessage [shamlet|You have to be logged in to do that!|]
31           redirect % AuthR.LoginR
32         Just userId -> do
```

# Local Definitions

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

```
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:

```
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.