```
-- setting the "warn-incomplete-patterns" flag asks GHC to warn you
-- about possible missing cases in pattern-matching definitions
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}

-- see https://wiki.haskell.org/Safe_Haskell
{-# LANGUAGE Safe #-}

module Assessed1 (checksum , step , bin2Bool , bool2Bin , notBin , andBin , orBin ,
deMorg1 , deMorg2 , deMorg3 , deMorg4 , equals , roots) where

import Types

--------------------------------------------------------------------------------
---------------- DO **NOT** MAKE ANY CHANGES ABOVE THIS LINE -------------------
--------------------------------------------------------------------------------
import Data.Int

{- Question 1 -}




checksum :: Integral a => [a] -> Bool
checksum numList = if length numList == 8 && sum numList `rem` 11 == 0 then True
                   else False

{- Question 2 -}

step :: Grid -> Grid
step [] = []
step g =
     [((x,y), col) | x <- [minX-1 .. maxX+1],
                             y <- [minY-1 .. maxY+1],
                             col <- [returnCorrectColour g (x, y)],
                (isDead (x,y) g && length (liveNeighbours g (x,y)) == 3)
             || (isLive (x,y) g && length (liveNeighbours g (x,y)) `elem` [2,3])
           ]
  where
    minX = minimum [ x | ((x,y), col) <- g ]
    maxX = maximum [ x | ((x,y), col) <- g ]
    minY = minimum [ y | ((x,y), col) <- g ]
    maxY = maximum [ y | ((x,y), col) <- g ]

getColours :: Grid -> [Coordinate] -> [Colour]
getColours g [] = []
getColours g coords = [colourOf g (coords !! i) | i <- [0 .. (length coords) - 1]]

returnCorrectColour :: Grid -> Coordinate -> Colour
returnCorrectColour g coord | isDead coord g     = blend (getColours g
(liveNeighbours g (coord)))
                            | otherwise         = colourOf g (coord)

-- The other Game of Life functions are in Types.hs to keep this file clean.
-- But life depends on step, so it needs to be here.
life :: Grid -> IO ()
life seed = f 0 seed
 where
  f n g = do
           terminalRender g
```

```
            putStrLn (show n)
            delayTenthSec 1
            f (n+1) (step g)

{- Question 3 -}
bin2Bool :: Binary -> Bool
bin2Bool Zero = False
bin2Bool One = True

bool2Bin :: Bool -> Binary
bool2Bin False = Zero
bool2Bin True = One

notBin :: Binary -> Binary
notBin Zero = One
notBin One = Zero

andBin :: Binary -> Binary -> Binary
andBin One One = One
andBin _ _ = Zero

orBin :: Binary -> Binary -> Binary
orBin One _ = One
orBin _ One = One
orBin Zero Zero = Zero

deMorg1 :: Binary -> Binary -> Binary
deMorg1 Zero Zero = One
deMorg1 _ _ = Zero

deMorg2 :: Binary -> Binary -> Binary
deMorg2 Zero Zero = One
deMorg2 _ _ = Zero

deMorg3 :: Binary -> Binary -> Binary
deMorg3  One One = Zero
deMorg3 _ _ = One

deMorg4 :: Binary -> Binary -> Binary
deMorg4  One One = Zero
deMorg4 _ _ = One




{- Question 4 -}
equals :: (Finite a, Eq b) => (a -> b) -> (a -> b) -> Bool
equals f g = and [f x == g x | x <- [minX..maxX]]
--equals = undefined
  where
    minX :: Bounded a => a
    minX = minBound
    maxX :: Bounded a => a
    maxX = maxBound

{- Question 5 -}
roots :: (Finite a , Num b, Eq b) => (a -> b) -> [a]
roots f = [x | x <- [minX..maxX], f x == 0]
  where
```

```
minX :: Bounded a => a
minX = minBound
maxX :: Bounded a => a
maxX = maxBound
```