



learn you a haskell  
in 20 minutes



Hello!

# Who am I?

- Student at UGent
- Geek
- I like to make things

@jaspervdj

jaspervdj.be

github.com/jaspervdj

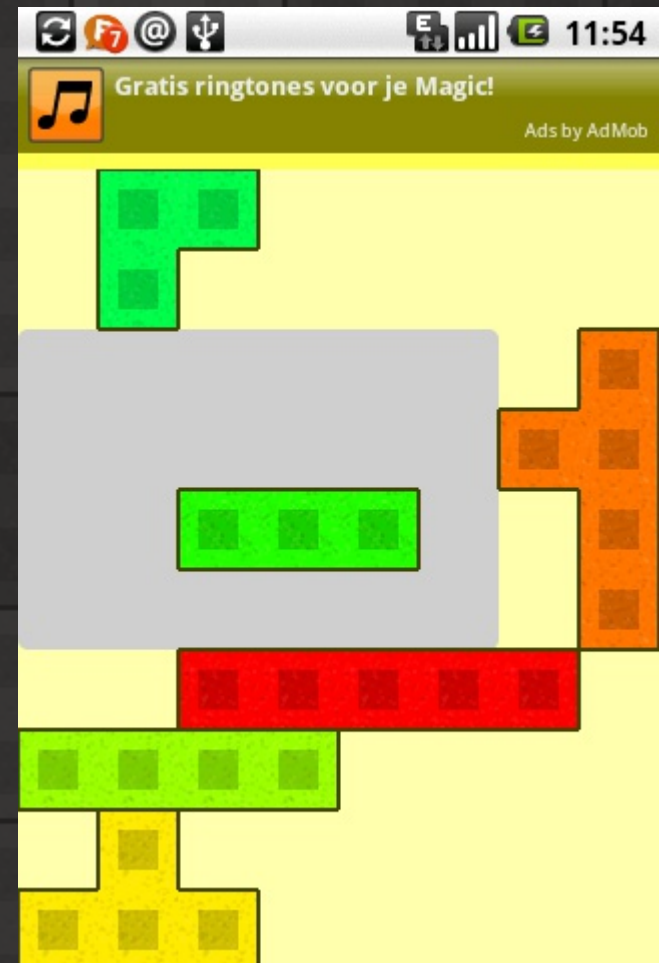
# What the hell is Haskell?

- Programming language
- Functional
- Based on  $\lambda$ -calculus

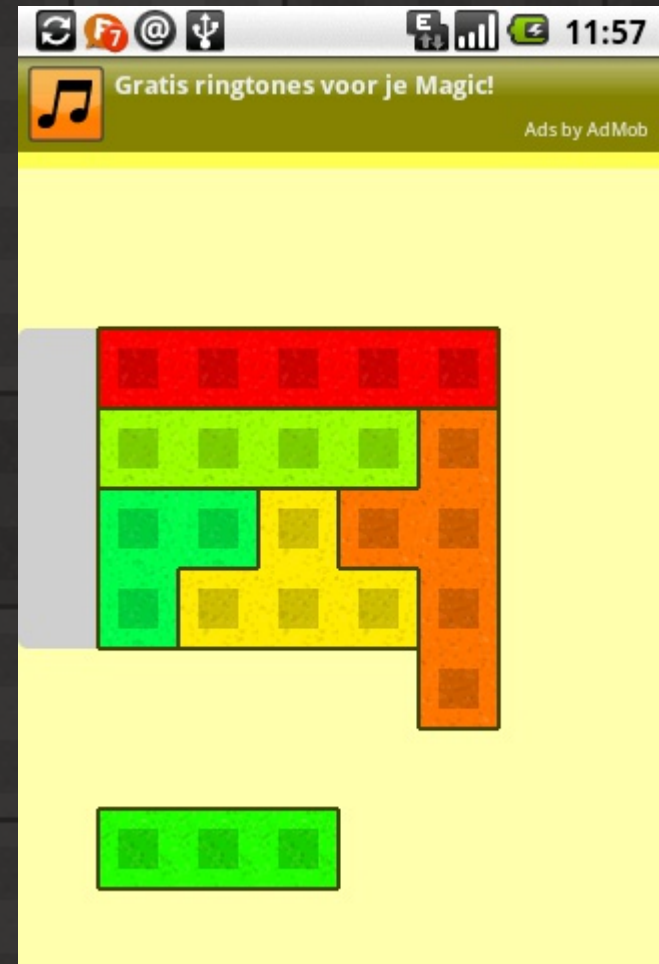
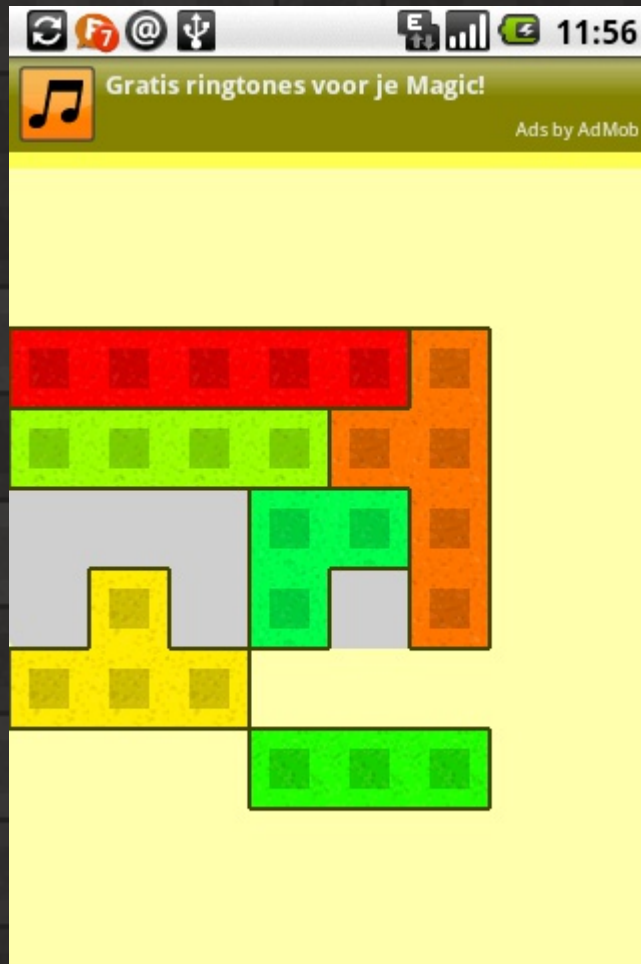
On to serious stuff!



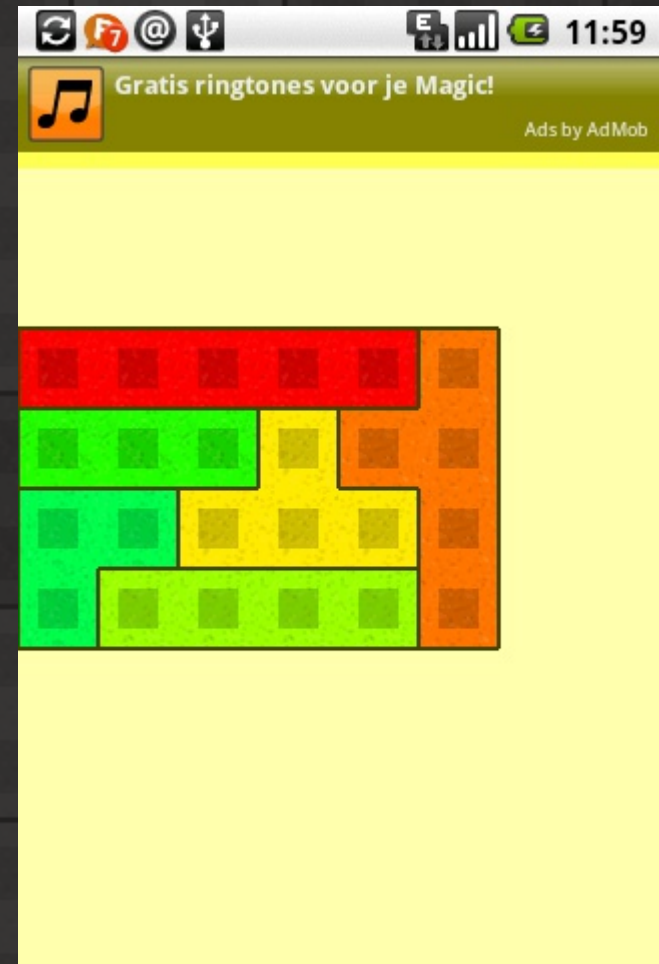
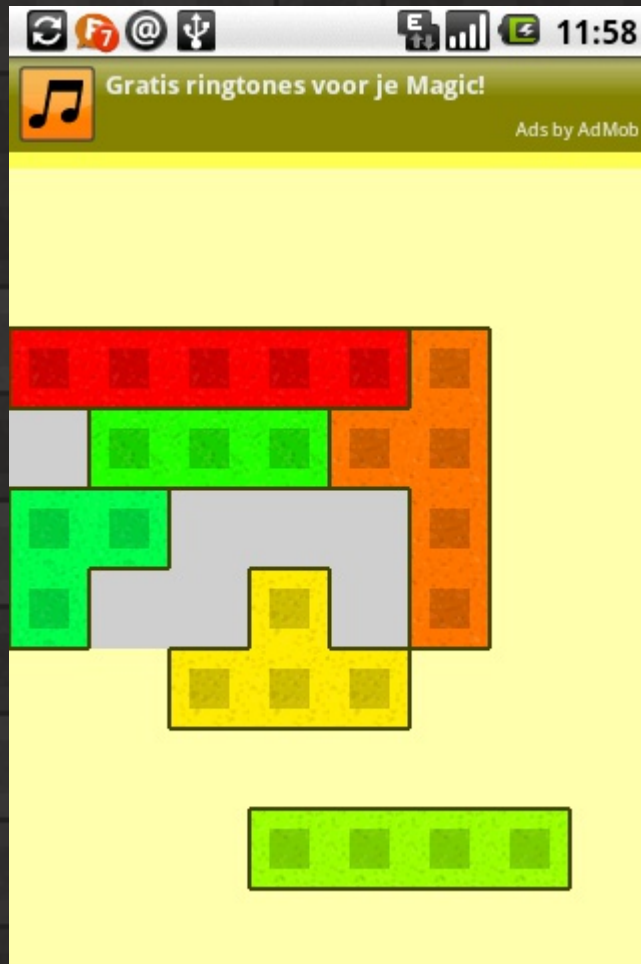
# I have this game on my phone...



... which leads to frustrations ....



... until you solve it ...





Yay!

Only about a thousand more  
puzzles to solve!

Yay!

Only about a thousand more  
puzzles to solve!

\*facepalm\*

# Simple data types

```
-- Let's represent tiles as simple points...  
data Tile = Tile { tileX :: Int  
                  , tileY :: Int  
                  } deriving (Eq, Ord, Show)
```

# Alias types

```
-- Let's represent tiles as simple points...
data Tile = Tile { tileX :: Int
                  , tileY :: Int
                  } deriving (Eq, Ord, Show)

-- A piece is basically a list of tiles.
type Piece = [Tile]
```

# Alias types

```
-- Let's represent tiles as simple points...  
data Tile = Tile { tileX :: Int  
                  , tileY :: Int  
                  } deriving (Eq, Ord, Show)  
  
-- A piece is basically a list of tiles.  
type Piece = [Tile]  
  
-- A board is basically a large piece.  
type Board = Piece
```

# Alias types

```
-- Let's represent tiles as simple points...  
data Tile = Tile { tileX :: Int  
                  , tileY :: Int  
                  } deriving (Eq, Ord, Show)
```

```
-- A piece is basically a list of tiles.
```

```
type Piece = [Tile]
```

```
-- A board is basically a large piece.
```

```
type Board = Piece
```

```
-- A solution is a list of pieces.
```

```
type Solution = [Piece]
```



**Haskell:**  
Putting the funk in  
funktion since 1990

# Simple functions

```
-- Determine the width of a piece.  
width :: Piece -> Int
```



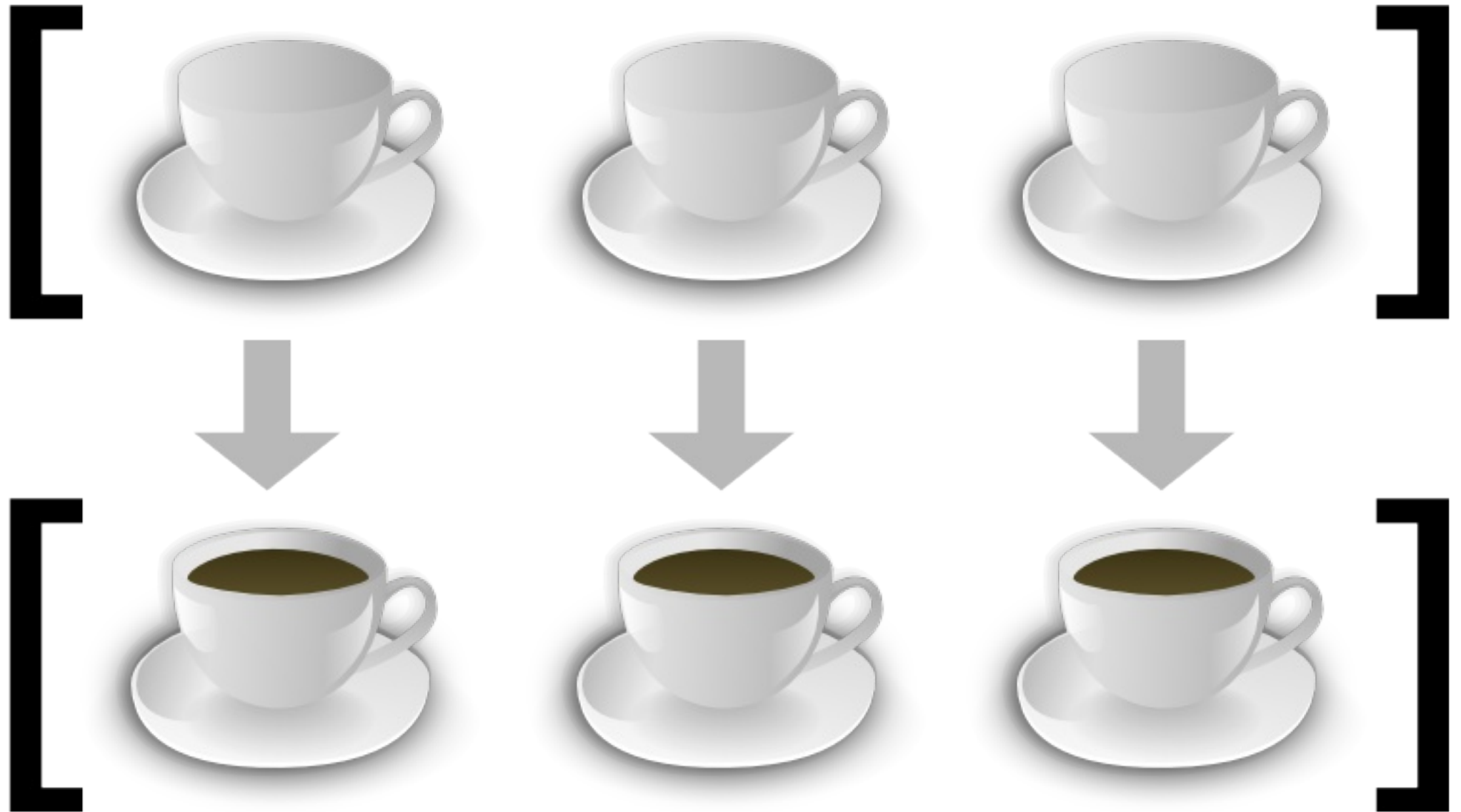
# Simple functions

```
-- Determine the width of a piece.  
width :: Piece -> Int  
width piece = maximum xs
```

# Simple functions

```
-- Determine the width of a piece.  
width :: Piece -> Int  
width piece = maximum xs  
  where xs = map tileX piece
```

map fill cups



# Simple functions

```
-- Determine the width of a piece.
```

```
width :: Piece -> Int
```

```
width piece = maximum xs
```

```
  where xs = map tileX piece
```

```
-- Determine the height of a piece.
```

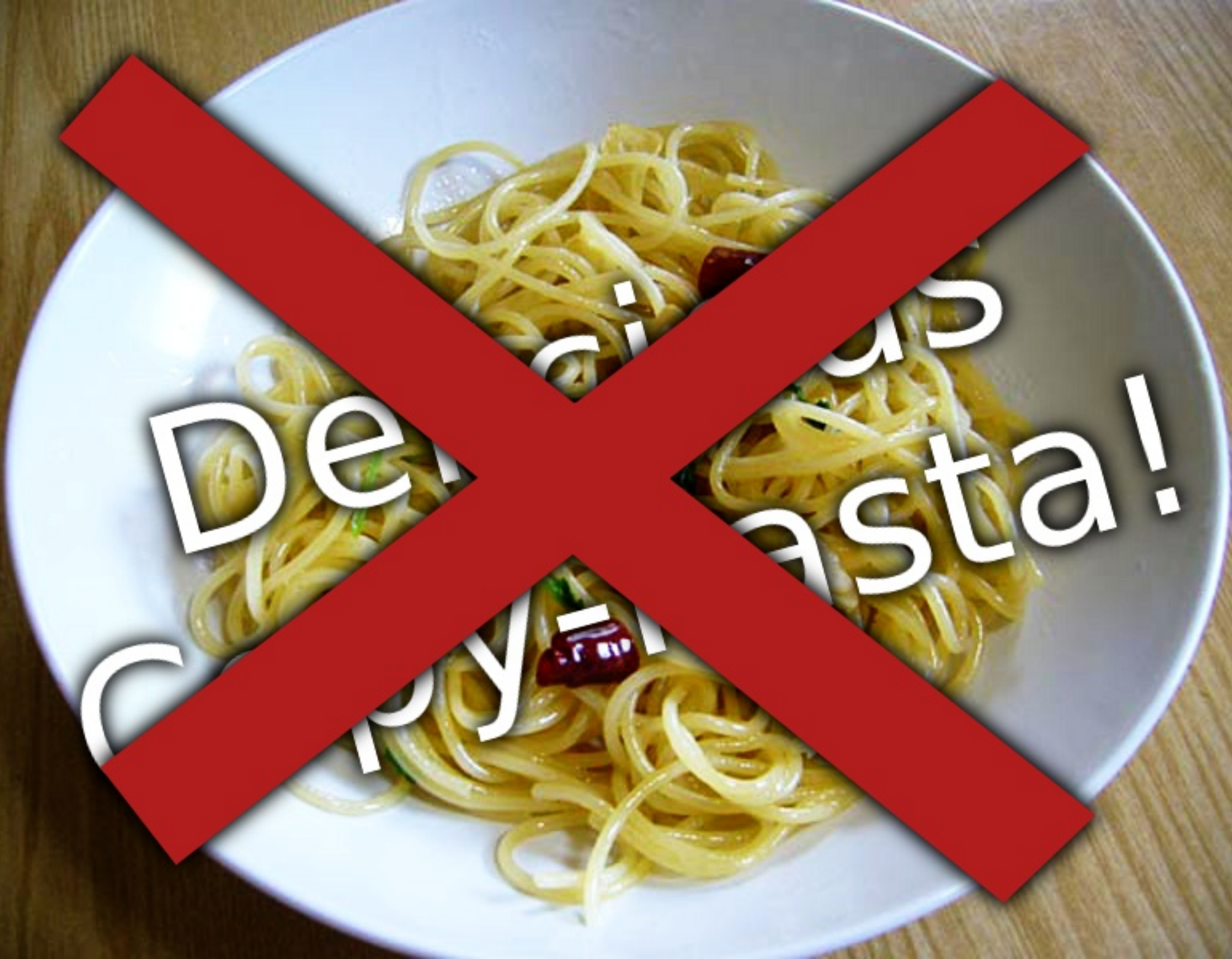
```
height :: Piece -> Int
```

```
height piece = maximum ys
```

```
  where ys = map tileY piece
```



Delicious  
Copy-Pasta!



De

ci

as

asta!

C

roy



Haskell likes it

**HOT**

(higher-order types)

# Higher-order functions

```
-- General dimension function.  
dimension :: (Piece -> Int) -> Piece -> Int  
dimension f piece = maximum xs  
  where xs = map f piece
```

```
-- Determine the width of a piece.
```

```
width :: Piece -> Int  
width = dimension tileX
```

```
-- Determine the height of a piece.
```


```
height :: Piece -> Int  
height = dimension tileY
```



# More utility functions

-- Move a piece.

```
translate :: Piece -> (Int, Int) -> Piece  
translate piece (x, y) =
```

A black and white photograph of a person wearing a Guy Fawkes mask, a symbol associated with the hacktivist group Anonymous. The person is holding a large, hand-drawn sign that reads "We Are Anonymous (functions)". The scene is set in a public square or street with historic buildings and other people in the background. The sign is made of cardboard and has a rough, hand-drawn appearance. The text is written in a simple, bold font. The person is wearing a light-colored t-shirt and dark pants. The background shows a cobblestone street and several multi-story buildings with many windows. There are other people walking around, some of whom are also wearing masks. The overall atmosphere is one of a public demonstration or protest.

We Are  
Anonymous  
(functions)

# More utility functions

```
-- Move a piece.
```

```
translate :: Piece -> (Int, Int) -> Piece
```

```
translate piece (x, y) =
```

```
  map \(Tile tx ty) ->
```

```
    Tile (tx + x) (ty + y)) piece
```

# More utility functions

-- Move a piece.

```
translate :: Piece -> (Int, Int) -> Piece
```

```
translate piece (x, y) =
```

```
  map \(Tile tx ty) ->
```

```
    Tile (tx + x) (ty + y)) piece
```

-- Produce a new board when putting a piece.

```
putPiece :: Board -> Piece -> Board
```

# More utility functions

-- Move a piece.

```
translate :: Piece -> (Int, Int) -> Piece
```

```
translate piece (x, y) =
```

```
  map \(Tile tx ty) ->
```

```
    Tile (tx + x) (ty + y)) piece
```

-- Produce a new board when putting a piece.

```
putPiece :: Board -> Piece -> Board
```

```
putPiece board piece = board \\ piece
```

# More utility functions

-- Move a piece.

```
translate :: Piece -> (Int, Int) -> Piece
```

```
translate piece (x, y) =
```

```
  map \(Tile tx ty) ->
```

```
    Tile (tx + x) (ty + y)) piece
```

-- Produce a new board when putting a piece.

```
putPiece :: Board -> Piece -> Board
```

```
putPiece board piece = board \\ piece
```

-- Check if we can put a piece on the board.

```
canPutPiece :: Board -> Piece -> Bool
```

# More utility functions

-- Move a piece.

```
translate :: Piece -> (Int, Int) -> Piece
```

```
translate piece (x, y) =
```

```
  map \(Tile tx ty) ->
```

```
    Tile (tx + x) (ty + y)) piece
```

-- Produce a new board when putting a piece.

```
putPiece :: Board -> Piece -> Board
```

```
putPiece board piece = board \\ piece
```

-- Check if we can put a piece on the board.

```
canPutPiece :: Board -> Piece -> Bool
```

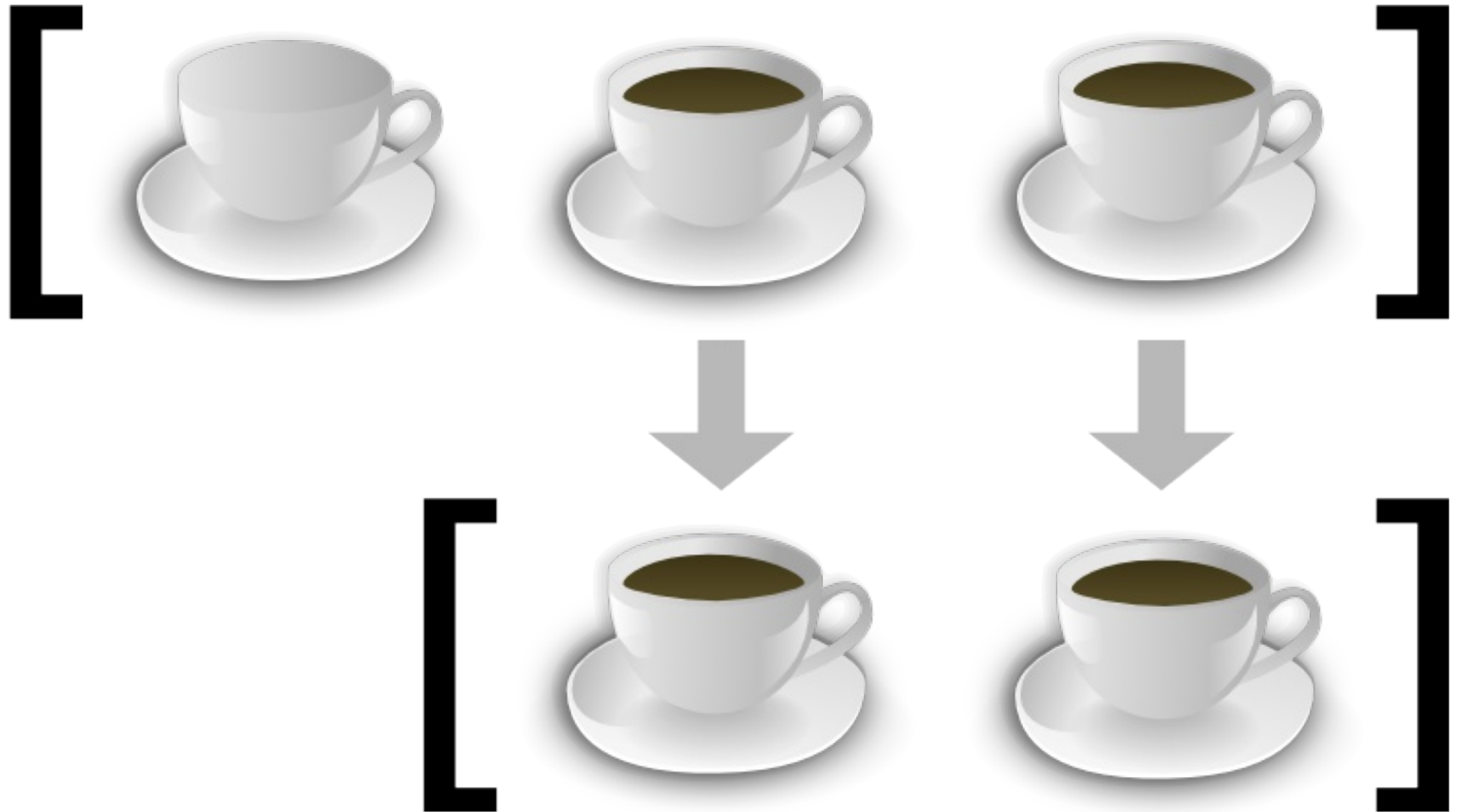
```
canPutPiece board = all ('elem' board)
```

# Validation of positions

```
validPositions :: Board -> Piece -> [Piece]
validPositions board piece =
    filter (canPutPiece board) allPositions
```



filter isFull cups



# Validation of positions

```
validPositions :: Board -> Piece -> [Piece]
validPositions board piece =
    filter (canPutPiece board) allPositions
  where allPositions :: [Piece]
        allPositions = map
            (translate piece) allCoords
```

# Validation of positions

```
validPositions :: Board -> Piece -> [Piece]
validPositions board piece =
  filter (canPutPiece board) allPositions
  where allPositions :: [Piece]
        allPositions = map
          (translate piece) allCoords
  -- A simple list with all coords.
  allCoords :: [(Int, Int)]
  allCoords = [(x, y) |
    x <- [0 .. (width board) - 1],
    y <- [0 .. (height board) - 1]]
```

# And finally, on to solving!

```
solve :: Board -> [Piece] ->  
      [Piece] -> Maybe Solution
```



# And finally, on to solving!

```
solve :: Board -> [Piece] ->  
      [Piece] -> Maybe Solution  
solve board piecesLeft added  
  | null piecesLeft = Just added
```

# And finally, on to solving!

```
solve :: Board -> [Piece] ->
      [Piece] -> Maybe Solution
solve board piecesLeft added
| null piecesLeft = Just added
| otherwise = msum solutions
```

# And finally, on to solving!

```
solve :: Board -> [Piece] ->
      [Piece] -> Maybe Solution
solve board piecesLeft added
| null piecesLeft = Just added
| otherwise = msum solutions
where solutions = map solve' positions
```



# And finally, on to solving!

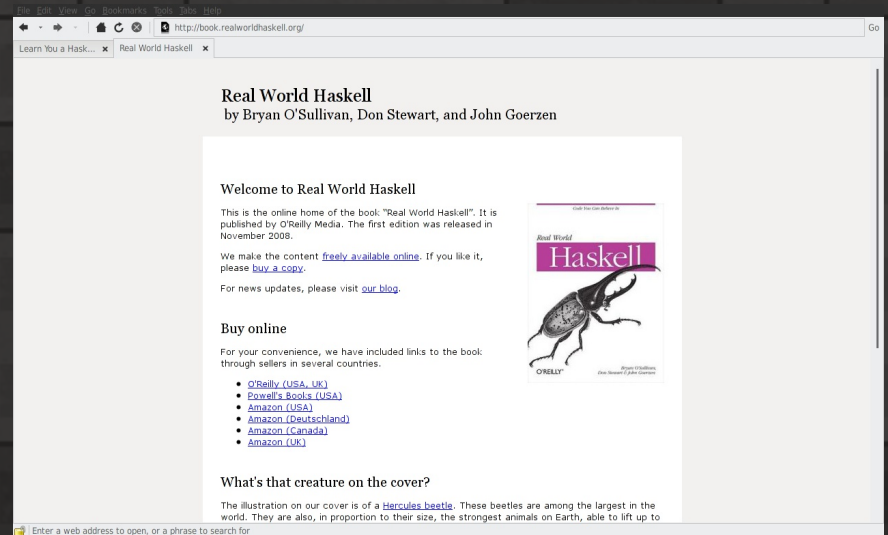
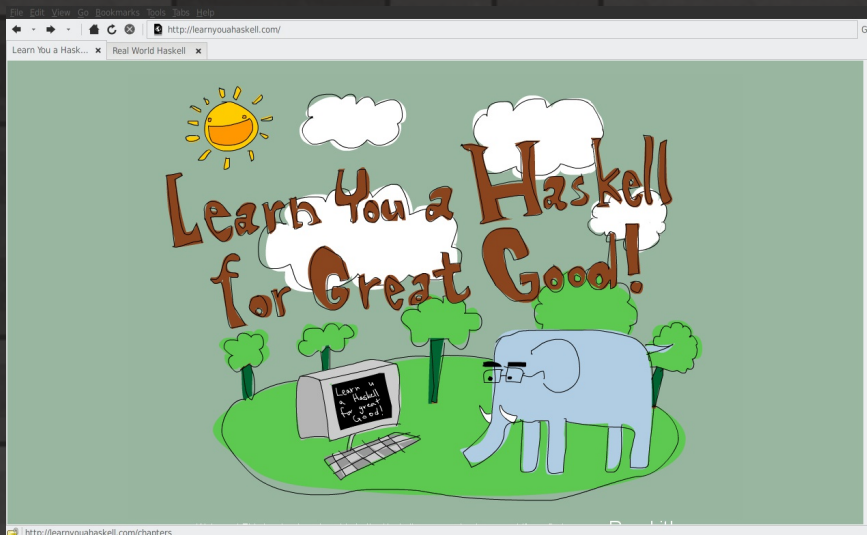
```
solve :: Board -> [Piece] ->
      [Piece] -> Maybe Solution
solve board piecesLeft added
| null piecesLeft = Just added
| otherwise = msum solutions
where solutions = map solve' positions
      positions = validPositions board
      (head piecesLeft)
```

# And finally, on to solving!

```
solve :: Board -> [Piece] ->
      [Piece] -> Maybe Solution
solve board piecesLeft added
| null piecesLeft = Just added
| otherwise = msum solutions
where solutions = map solve' positions
      positions = validPositions board
      (head piecesLeft)
      solve' piece = solve
      (putPiece board piece)
      (tail piecesLeft) (piece:added)
```

# Many good resources available online!

learnyouahaskell.com  
book.realworldhaskell.org





Questions?  
Demo?