learn you a haskell in 20 minutes

Hello!

Who am I?

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

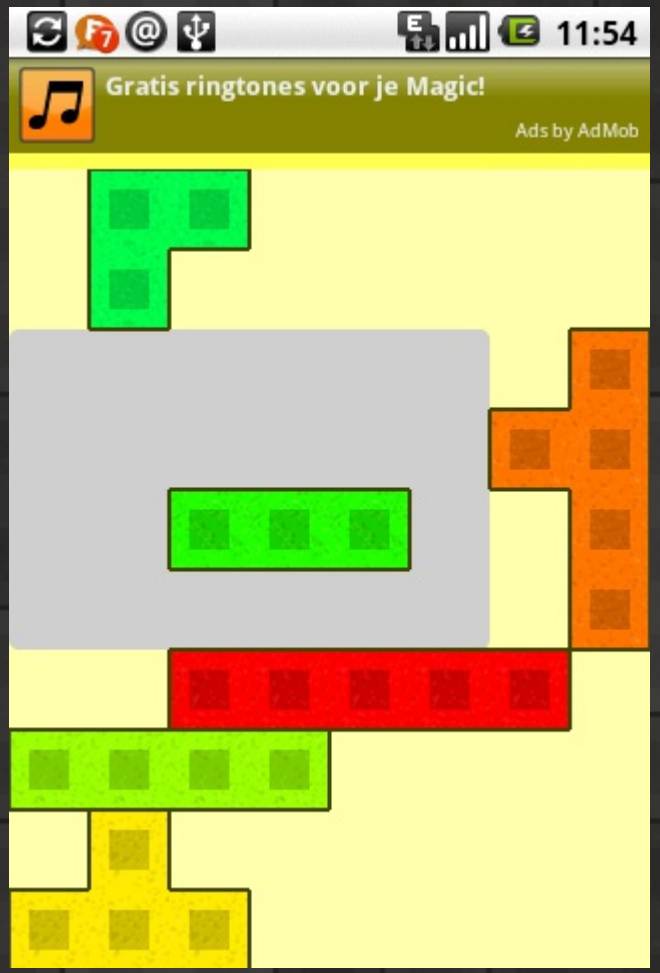
@jaspervdj
jaspervdj.be
github.com/jaspervdj

What the hell is Haskell?

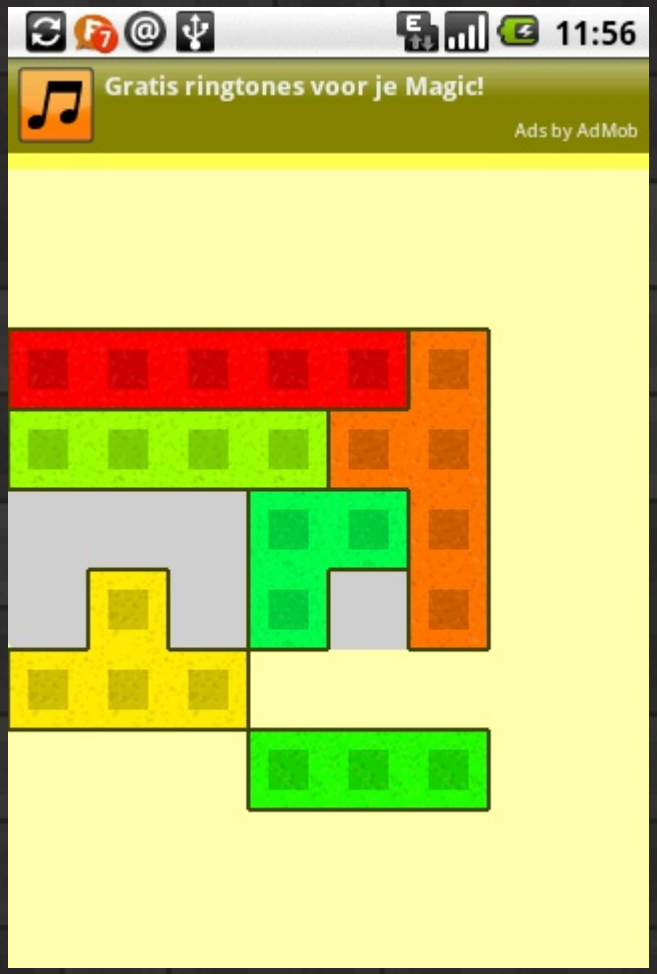
- Programming language
- Functional
- Based on λ-calculus

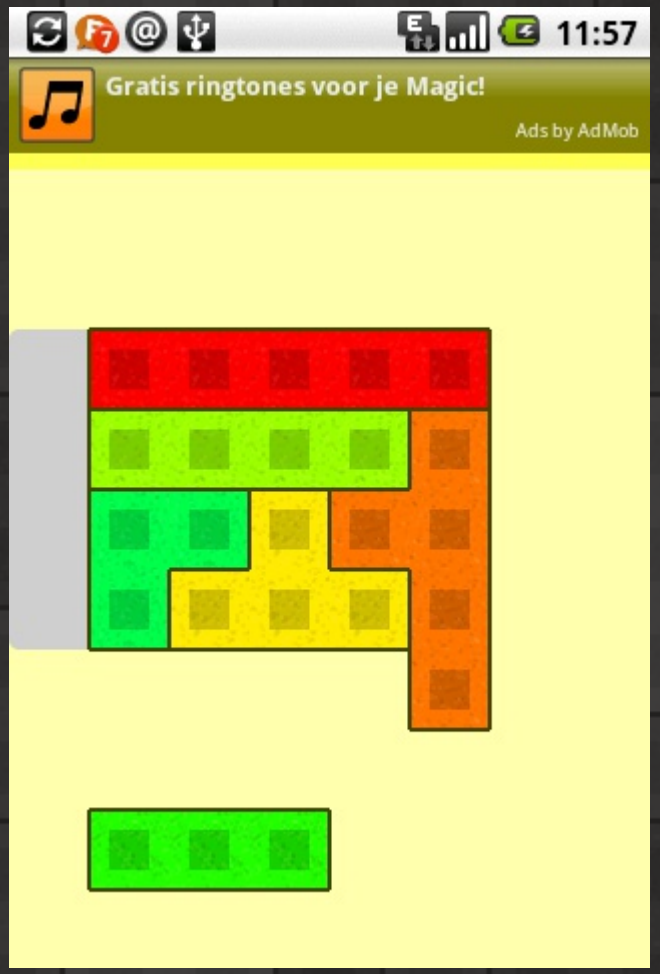
On to serious stuff!

I have this game on my phone...

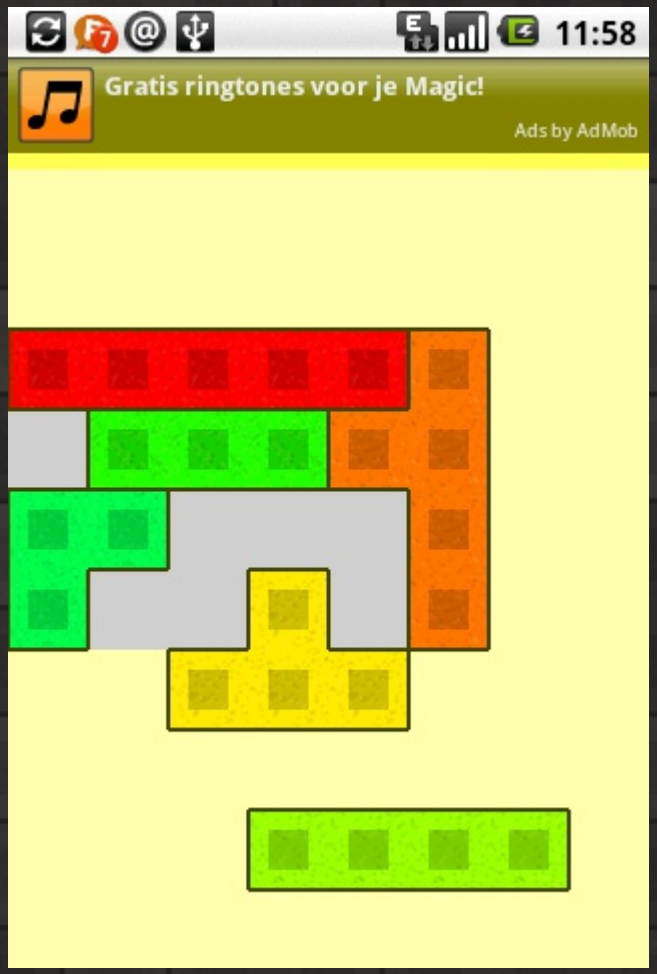


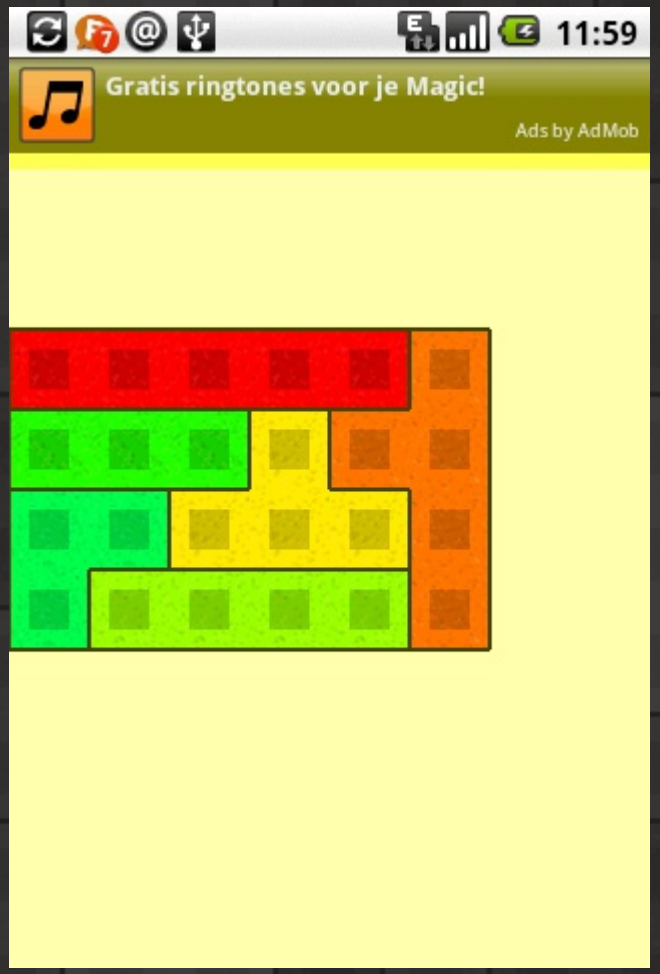
... which leads to frustrations ....





... 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

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

Alias types

```haskell
-- 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

```haskell
-- 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
```

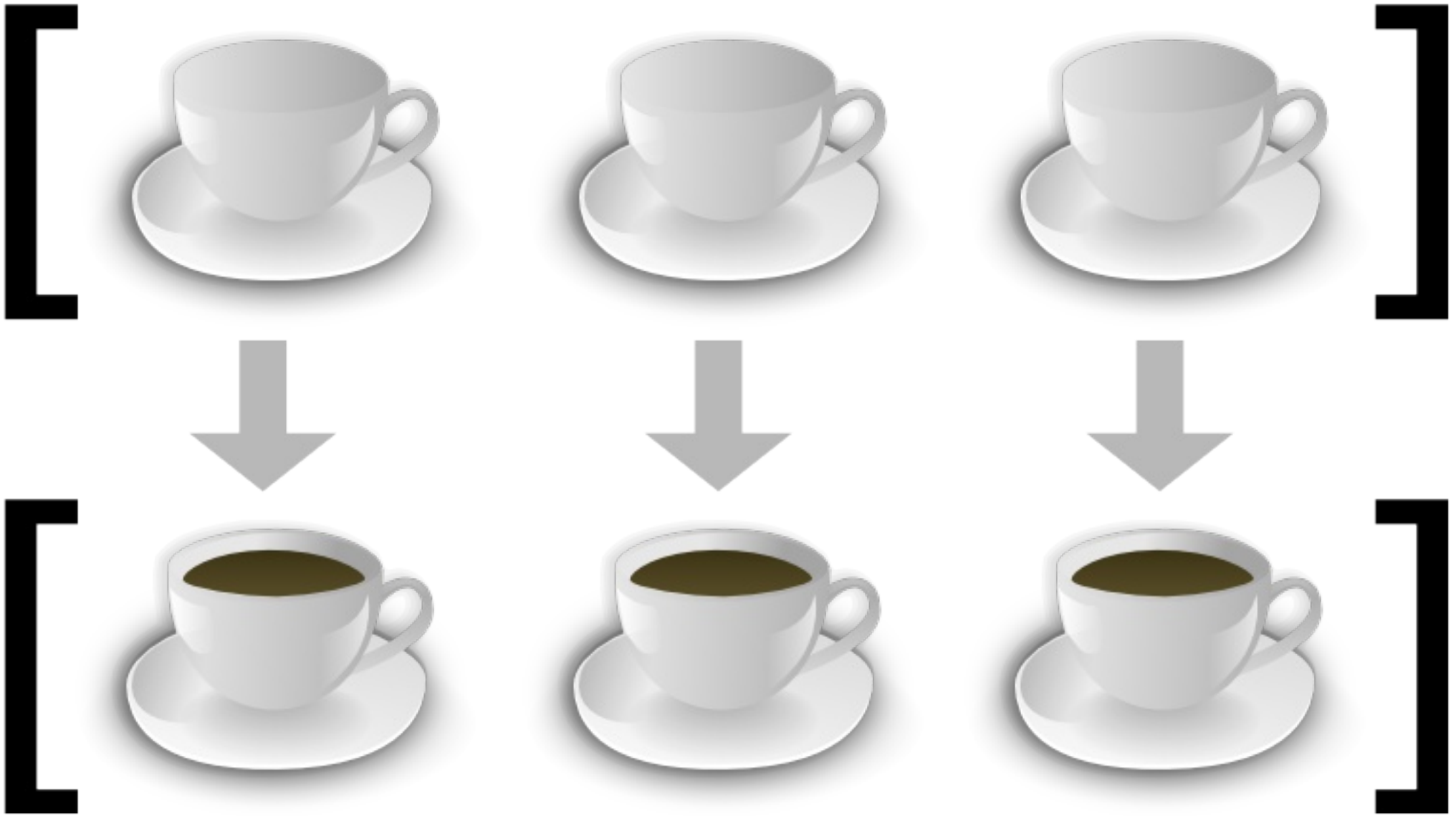
Simple functions

```
-- 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

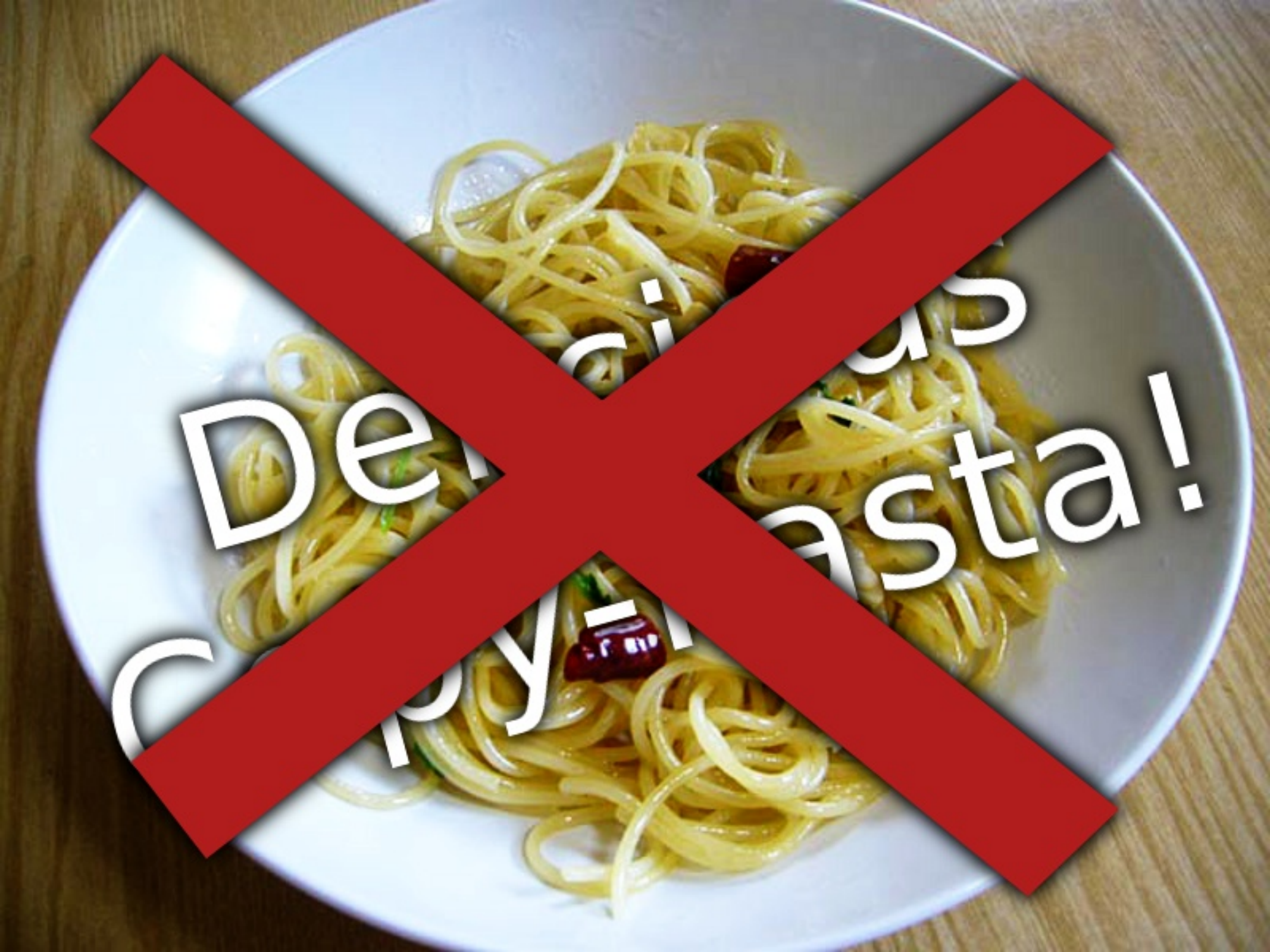


Simple functions

```haskell
-- 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
asta!
Copy-P


Haskell likes it
HOT
(higher-order types)
λ

Higher-order functions

```haskell
-- 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) =
```


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
```

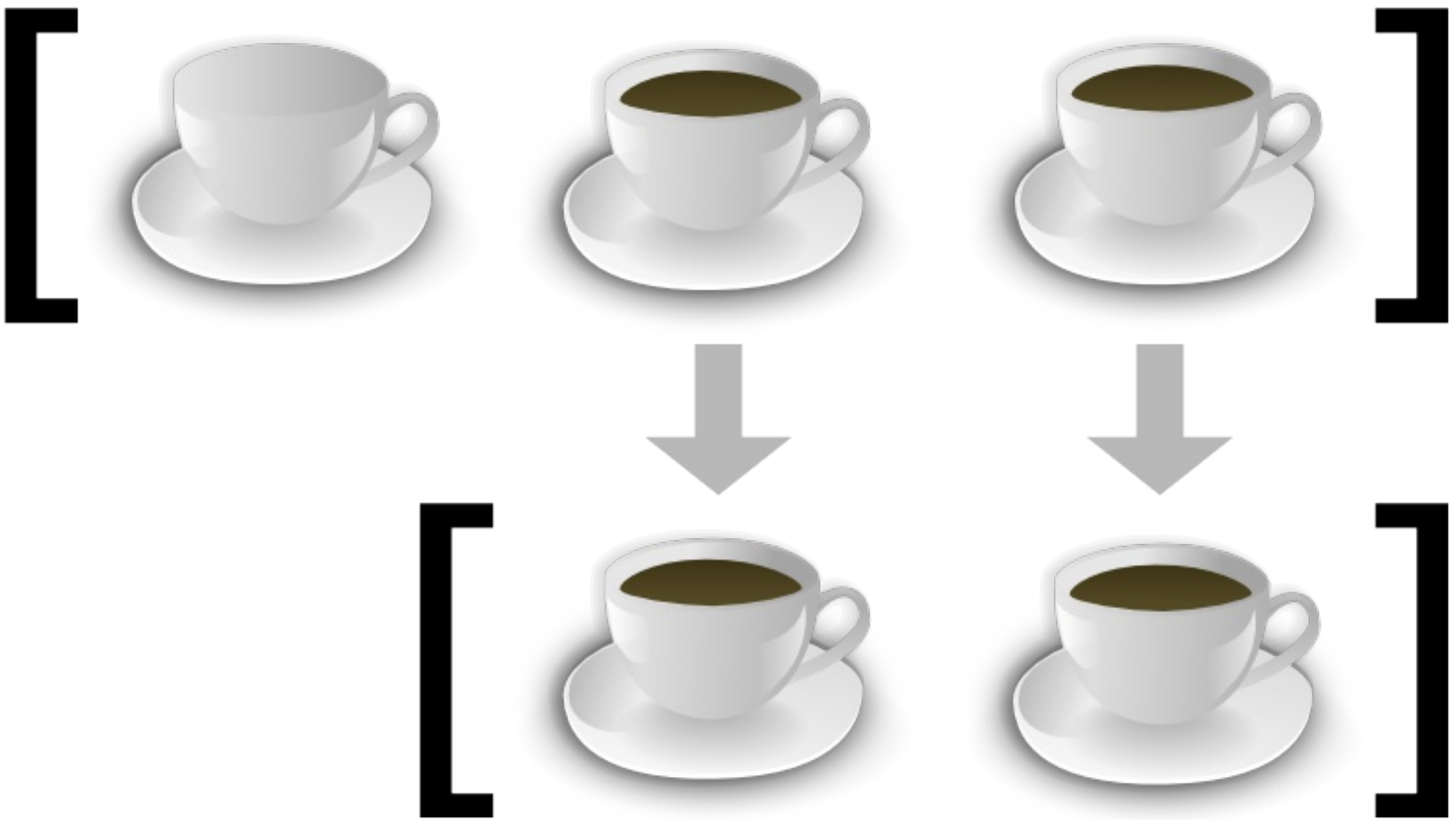
More utility functions

```
-- 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
```


λ

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!

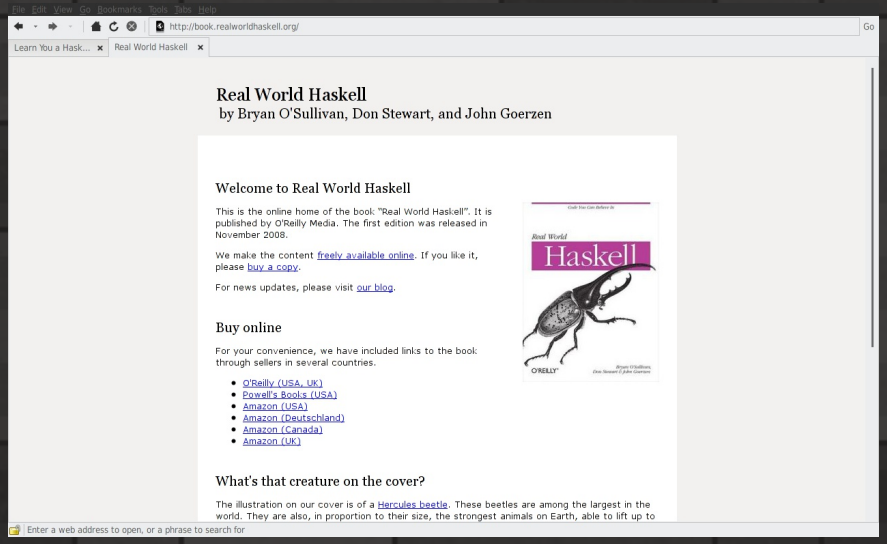
```
solve :: Board -> [Piece] ->
         [Piece] -> Maybe Solution
solve board piecesLeft added
  | null piecesLeft = Just added
  | otherwise = msum solutions
 where solutions = map solve' positions
       positions = validPositions board
                   (head piecesLeft)
```

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?