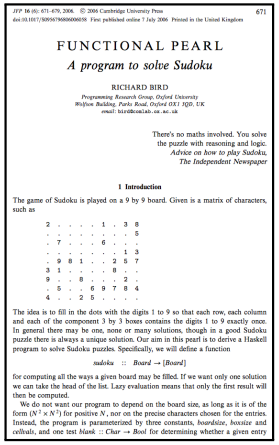
Coding dojo
A simple Sudoku solver in Haskell

B◦FP

Bucharest FP #027

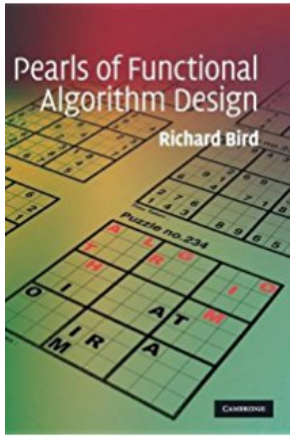
Welcome

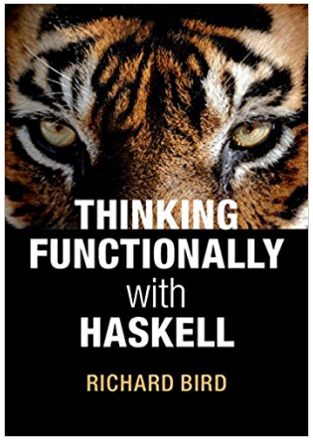


FUNCTIONAL PEARL

A program to solve Sudoku

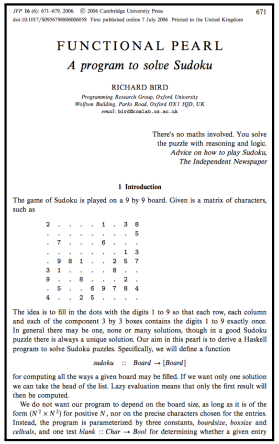
RICHARD BIRD





- Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)

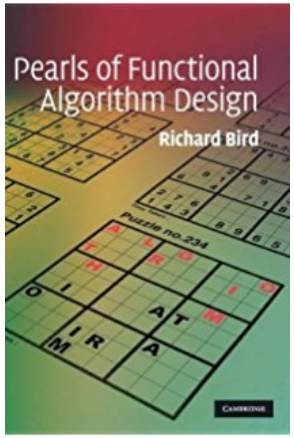
Welcome

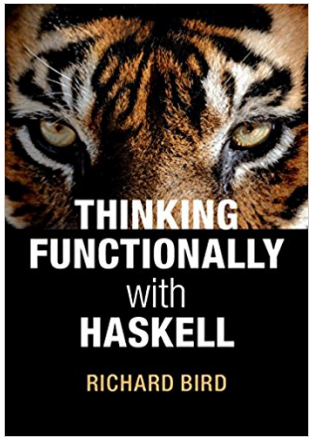


FUNCTIONAL PEARL

A program to solve Sudoku

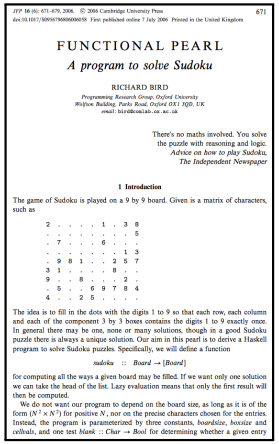
RICHARD BIRD





- Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)
- Twelve short functions:
  - 5 easy ($\star$)
  - 4 medium ($\star\star$)
  - 3 challenging ($\star\star\star$)

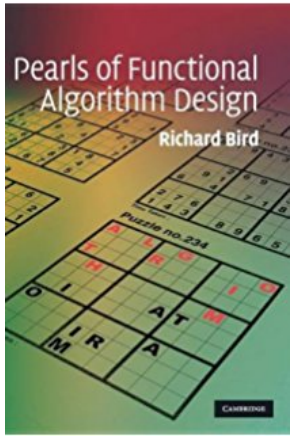
Welcome

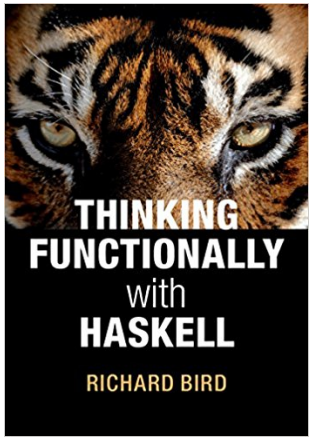


FUNCTIONAL PEARL

A program to solve Sudoku

RICHARD BIRD





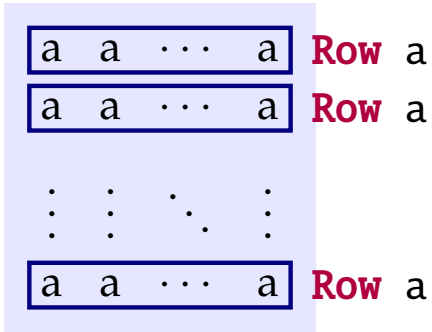
- Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)
- Twelve short functions:
  - 5 easy ($\star$)
  - 4 medium ($\star\star$)
  - 3 challenging ($\star\star\star$)
- Top-down programming / wishful thinking

How to play Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

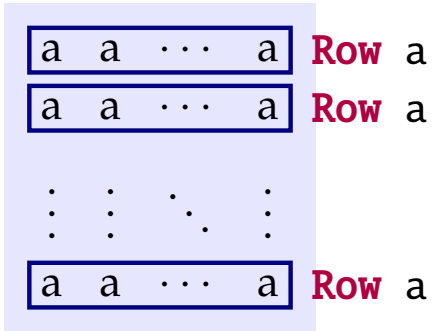
Fill in the empty cells with digits 1 to N^2 such that every row, column and $N \times N$ box contains the digits 1 to N^2.

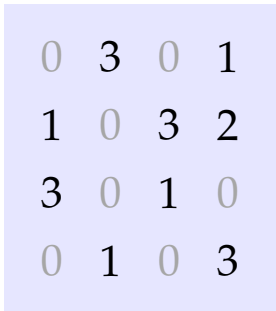
Data types




```
type Matrix a = [Row a]
type Row a = [a]
```

Data types

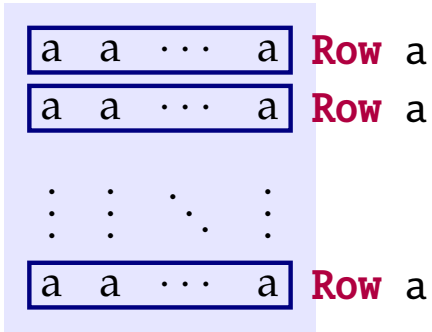


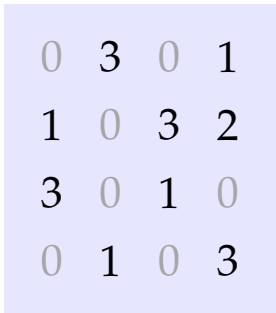



```
type Matrix a = [Row a]
type Row a = [a]

type Grid = Matrix Digit
type Digit = Int
```

Data types






```
type Matrix a = [Row a]
type Row a = [a]

type Grid = Matrix Digit
type Digit = Int
```

We assume that digit zero indicates an empty cell:

```
isEmpty :: Digit -> Bool
isEmpty 0 = True
isEmpty _ = False
```

Exercise 1: `solve` [⋆]

```haskell
solve :: Grid -> [Grid]
solve = undefined
```

Exercise 1: `solve` [⋆]

```haskell
solve :: Grid -> [Grid]
solve = undefined
```

Given:

```haskell
-- Generates grids by replacing empty entries
-- with all possible choices
completions :: Grid -> [Grid]
```

Exercise 1: `solve` [⋆]

```haskell
solve :: Grid -> [Grid]
solve = undefined
```

Given:

```haskell
-- Generates grids by replacing empty entries
-- with all possible choices
completions :: Grid -> [Grid]

-- Tests whether a grid is a valid solution:
-- has different entries in each row, column and box
valid :: Grid -> Bool
```

Exercise 1: `solve` [⋆]

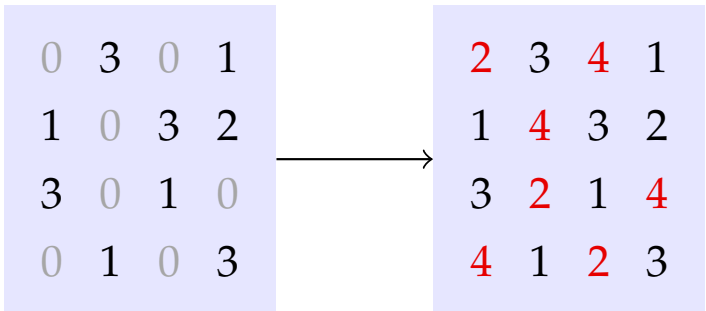
```haskell
solve :: Grid -> [Grid]
solve = undefined
```

Given:

```haskell
-- Generates grids by replacing empty entries
-- with all possible choices
completions :: Grid -> [Grid]

-- Tests whether a grid is a valid solution:
-- has different entries in each row, column and box
valid :: Grid -> Bool
```

Example:



`Grid` `Grid`

Exercise 2: `completions` [⋆]

```haskell
completions :: Grid -> [Grid]
completions = undefined
```

Exercise 2: `completions` [⋆]

```haskell
completions :: Grid -> [Grid]
completions = undefined
```

Given:

```haskell
-- Replaces empty entries with all possible choices
-- for that entry
choices :: Grid -> Matrix [Digit]
```

Exercise 2: `completions` [⋆]

```haskell
completions :: Grid -> [Grid]
completions = undefined
```

Given:

```haskell
-- Replaces empty entries with all possible choices
-- for that entry
choices :: Grid -> Matrix [Digit]

-- Generates a list of all possible boards
-- from a given matrix of choices
expand :: Matrix [Digit] -> [Grid]
```

Exercise 2: `completions` [⋆]

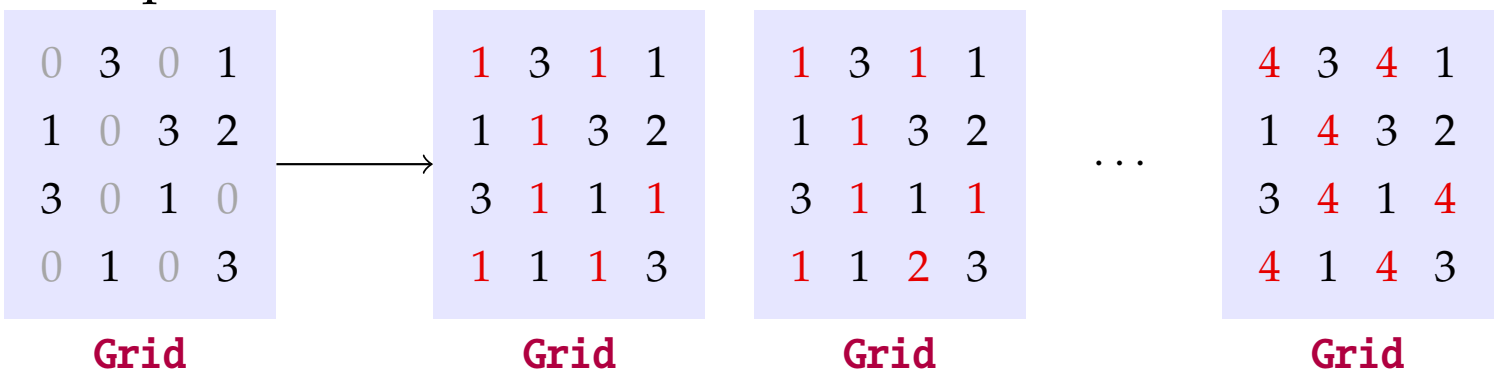
```haskell
completions :: Grid -> [Grid]
completions = undefined
```

Given:

```haskell
-- Replaces empty entries with all possible choices
-- for that entry
choices :: Grid -> Matrix [Digit]

-- Generates a list of all possible boards
-- from a given matrix of choices
expand :: Matrix [Digit] -> [Grid]
```

Example:



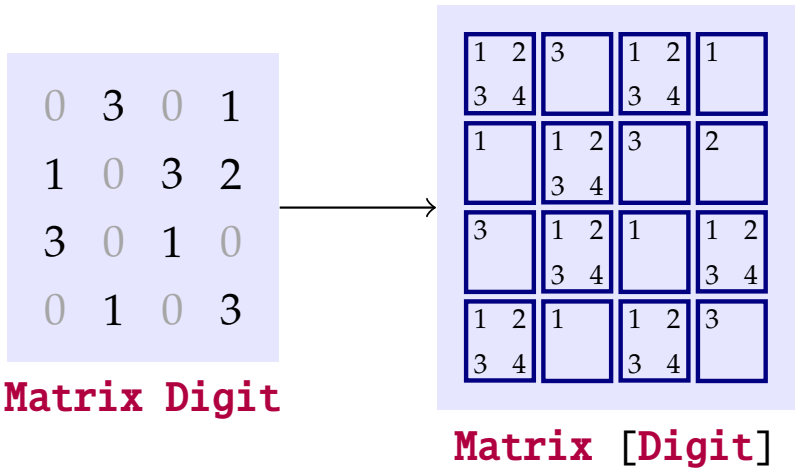
Exercise 3: `choices` $[\star]$

```
choices :: Grid -> Matrix [Digit]
choices = undefined
```

Exercise 3: `choices` [⋆]

```
choices :: Grid -> Matrix [Digit]
choices = undefined
```

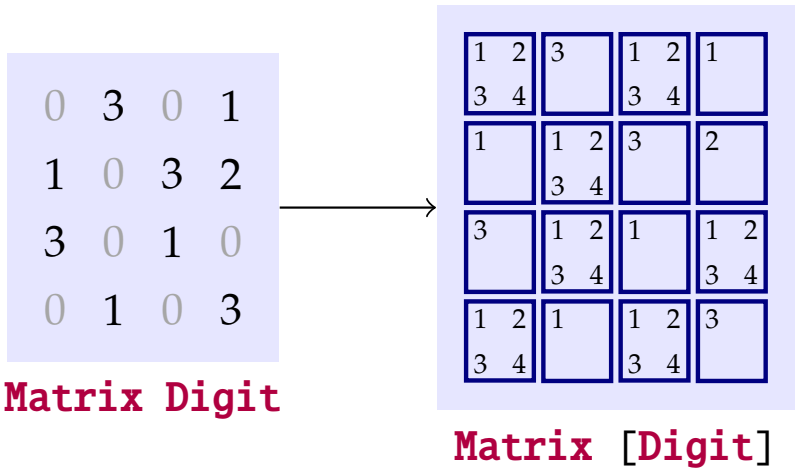
Example:



Exercise 3: `choices` [⋆]

```
choices :: Grid -> Matrix [Digit]
choices = undefined
```

Example:



Hint:

- Define a helper function `choice :: Digit -> [Digit]`

Exercise 4: expand [⋆⋆]

```haskell
expand :: Matrix [Digit] -> [Grid]
expand = undefined
```

Exercise 4: expand [⋆⋆]

```haskell
expand :: Matrix [Digit] -> [Grid]
expand = undefined
```

Given:

```haskell
-- Computes the cartesian product of a list of lists
cp :: [[a]] -> [[a]]
```

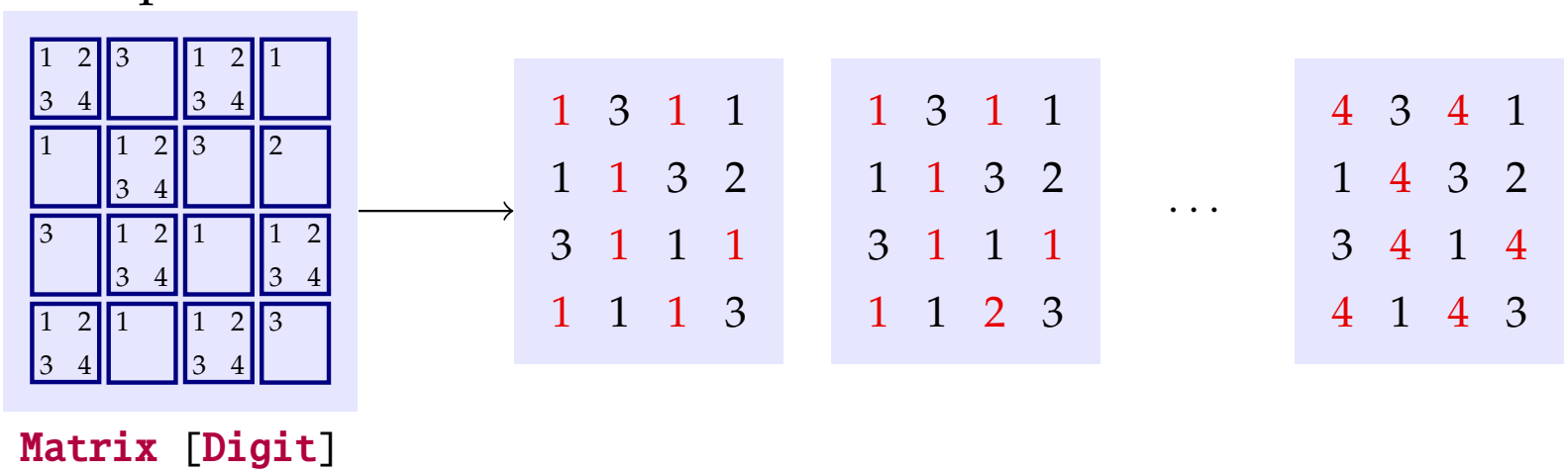
Exercise 4: expand [⋆⋆]

```
expand :: Matrix [Digit] -> [Grid]
expand = undefined
```

Given:

```
-- Computes the cartesian product of a list of lists
cp :: [[a]] -> [[a]]
```

Example:



`Matrix [Digit]`

Exercise 5: cp $[\star\star\star]$

```
cp :: [[a]] -> [[a]]
cp = undefined
```

Exercise 5: cp $[\star\star\star]$

```
cp :: [[a]] -> [[a]]
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Exercise 5: cp $[\star\star\star]$

```
cp :: [[a]] -> [[a]]
cp = undefined
```

Example:

```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion

Intermezzo

- Finished implementing the `completions` function
- Test `expand` and `completions` using the given examples

Exercise 6: `valid` [⋆⋆]

```
valid :: Grid -> Bool
valid = undefined
```

Exercise 6: `valid` [⋆⋆]

```haskell
valid :: Grid -> Bool
valid = undefined
```

Given:

```haskell
-- Checks that a list contains no duplicates
nodups :: [a] -> Bool
```

Exercise 6: `valid` [⋆⋆]

```haskell
valid :: Grid -> Bool
valid = undefined
```

Given:

```haskell
-- Checks that a list contains no duplicates
nodups :: [a] -> Bool

-- Re-orders the values from a matrix's rows, columns
-- or boxes to appear along the rows
rows :: Matrix a -> Matrix a
cols :: Matrix a -> Matrix a
boxs :: Matrix a -> Matrix a
```

Exercise 6: `valid` [⋆⋆]

```haskell
valid :: Grid -> Bool
valid = undefined
```

Given:

```haskell
-- Checks that a list contains no duplicates
nodups :: [a] -> Bool

-- Re-orders the values from a matrix's rows, columns
-- or boxes to appear along the rows
rows :: Matrix a -> Matrix a
cols :: Matrix a -> Matrix a
boxs :: Matrix a -> Matrix a
```

Examples:

$$\begin{matrix} 1 & 3 & 1 & 1 \\ 1 & 1 & 3 & 2 \\ 3 & 1 & 1 & 1 \\ 1 & 1 & 1 & 3 \end{matrix} \longrightarrow \texttt{False} \qquad \begin{matrix} 2 & 3 & 4 & 1 \\ 1 & 4 & 3 & 2 \\ 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 \end{matrix} \longrightarrow \texttt{True}$$

Exercise 7: nodups [⋆⋆]

```
nodups :: [a] -> Bool
nodups = undefined
```

Exercise 7: nodups [⋆⋆]

```
nodups :: [a] -> Bool
nodups = undefined
```

Examples:

```
nodups [] = True
nodups [1, 2, 3] = True
nodups [1, 2, 1] = False
```

Exercise 7: nodups [⋆⋆]

```
nodups :: [a] -> Bool
nodups = undefined
```

Examples:

```
nodups [] = True
nodups [1, 2, 3] = True
nodups [1, 2, 1] = False
```

Hints:

- Use recursion
- Use Hoogle to find a function of type `a -> [a] -> Bool`

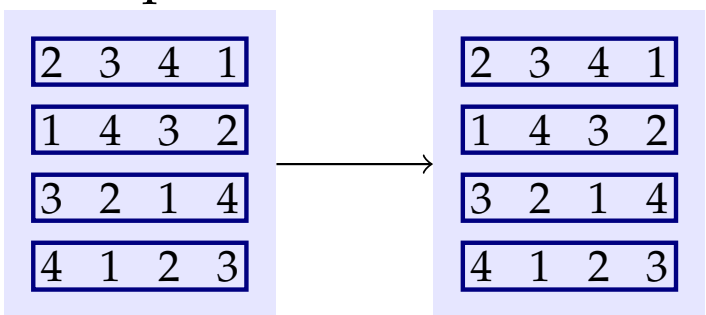
Exercise 8: `rows` $[\star]$

```
rows :: Matrix a -> Matrix a
```

Exercise 8: `rows` [⋆]

```
rows :: Matrix a -> Matrix a
```

Example:



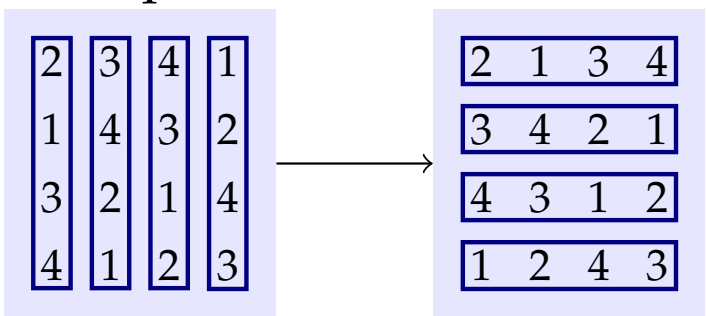
Exercise 9: `cols` $[\star\star\star]$

```haskell
cols :: Matrix a -> Matrix a
```

Exercise 9: `cols` [⋆⋆⋆]

```
cols :: Matrix a -> Matrix a
```

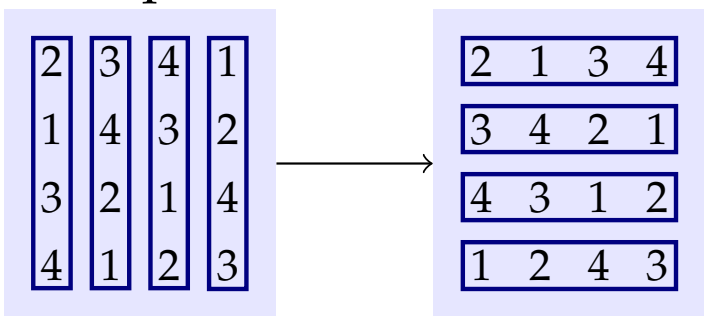
Example:



Exercise 9: `cols` $[\star\star\star]$

```
cols :: Matrix a -> Matrix a
```

Example:



Hints:

- Use recursion
- Use the `zipWith` function

Exercise 10: `boxs` $[\star\star\star]$

```
boxs :: Matrix a -> Matrix a
```

Exercise 10: `boxs` $[\star\star\star]$

```
boxs :: Matrix a -> Matrix a
```

Given:

```
-- Groups a list into lists of length two
group :: [a] -> [[a]]
```

Exercise 10: `boxs` $[\star\star\star]$

```
boxs :: Matrix a -> Matrix a
```

Given:

```
-- Groups a list into lists of length two
group :: [a] -> [[a]]

-- Flattens a nested list of elements
ungroup :: [[a]] -> [a]
```

Exercise 10: `boxs` [⋆⋆⋆]

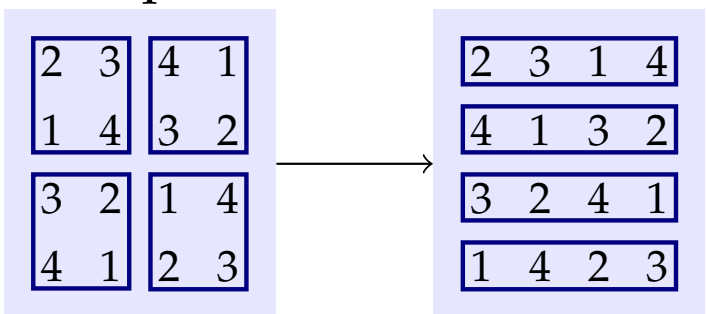
```
boxs :: Matrix a -> Matrix a
```

Given:

```
-- Groups a list into lists of length two
group :: [a] -> [[a]]

-- Flattens a nested list of elements
ungroup :: [[a]] -> [a]
```

Example:



Exercise 11: `group` [⋆⋆]

```haskell
group :: [a] -> [[a]]
```

Exercise 11: group [⋆⋆]

```
group :: [a] -> [[a]]
```

Example:

```
group [1,2,3,4] = [[1,2],[3,4]]
```

Exercise 12: ungroup $[\star]$

```
ungroup :: [[a]] -> [a]
```

Exercise 12: ungroup $[\star]$

```
ungroup :: [[a]] -> [a]
```

Example:

```
ungroup [[1,2],[3,4]] = [1,2,3,4]
```

Exercise 12: ungroup [⋆]

```
ungroup :: [[a]] -> [a]
```

Example:

```
ungroup [[1,2],[3,4]] = [1,2,3,4]
```

Hints:

- Use Hoogle

That's all folks

- Time to solve some Sudokus
- The current approach is very inefficient, but correct
- Next time: use equational reasoning to refactor and optimize

References

Bird, R. (2010). *Pearls of Functional Algorithm Design*. Cambridge University Press.

Bird, R. (2014). *Thinking Functionally with Haskell*. Cambridge University Press.

Bird, R. S. (2006). A program to solve Sudoku. *Journal of Functional Programming*, 16(6):671–679.