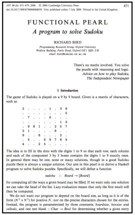
# Coding dojo
# A simple Sudoku solver in Haskell

B∘FP

Bucharest FP #027

# Welcome


JFP 36 (6): 671–679, 2006. © 2006 Cambridge University Press 671
doi:10.1017/S0956796806006058 First published online 7 July 2006 Printed in the United Kingdom

FUNCTIONAL PEARL

*A program to solve Sudoku*

RICHARD BIRD

*Programming Research Group, Oxford University*
*Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
*email:* bird@comlab.ox.ac.uk

There's no maths involved. You solve the puzzle with reasoning and logic.
*Advice on how to play Sudoku,*
*The Independent Newspaper*

**1 Introduction**

The game of Sudoku is played on a 9 by 9 board. Given is a matrix of characters, such as

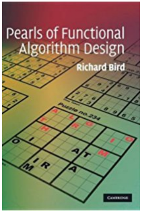
```
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

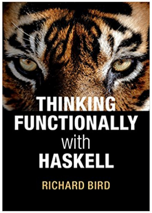
The idea is to fill in the dots with the digits 1 to 9 so that each row, each column and each of the component 3 by 3 boxes contains the digits 1 to 9 exactly once. In general there may be one, none or many solutions, though in a good Sudoku puzzle there is always a unique solution. Our aim in this pearl is to derive a Haskell program to solve Sudoku puzzles. Specifically, we will define a function

$$sudoku :: Board \to [Board]$$

for computing all the ways a given board may be filled. If we want only one solution we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

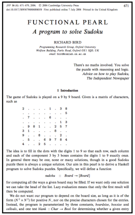
We do not want our program to depend on the board size, as long as it is of the form $(N^2 \times N^2)$ for positive $N$, nor on the precise characters chosen for the entries. Instead, the program is parameterized by three constants, *boardsize*, *boxsize* and *cellvals*, and one test $blank :: Char \to Bool$ for determining whether a given entry




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

# Welcome



JFP 16 (6): 671–679, 2006. © 2006 Cambridge University Press 671
doi:10.1017/S0956796806006058 First published online 7 July 2006 Printed in the United Kingdom

FUNCTIONAL PEARL

*A program to solve Sudoku*

RICHARD BIRD
*Programming Research Group, Oxford University*
*Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
email: bird@comlab.ox.ac.uk

There's no maths involved. You solve the puzzle with reasoning and logic.
*Advice on how to play Sudoku,*
*The Independent Newspaper*

**1 Introduction**

The game of Sudoku is played on a 9 by 9 board. Given is a matrix of characters, such as

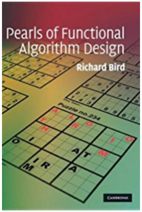
```
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

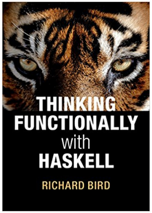
The idea is to fill in the dots with the digits 1 to 9 so that each row, each column and each of the component 3 by 3 boxes contains the digits 1 to 9 exactly once. In general there may be one, none or many solutions, though in a good Sudoku puzzle there is always a unique solution. Our aim in this pearl is to derive a Haskell program to solve Sudoku puzzles. Specifically, we will define a function

$$sudoku \;::\; Board \rightarrow [Board]$$

for computing all the ways a given board may be filled. If we want only one solution we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

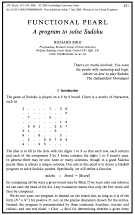
We do not want our program to depend on the board size, as long as it is of the form $(N^2 \times N^2)$ for positive $N$, nor on the precise characters chosen for the entries. Instead, the program is parameterized by three constants, *boardsize*, *boxsize* and *cellvals*, and one test $blank :: Char \rightarrow Bool$ for determining whether a given entry




- 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


JFP 36 (6): 671–679, 2006. © 2006 Cambridge University Press 671
doi:10.1017/S0956796806006058 First published online 7 July 2006 Printed in the United Kingdom

FUNCTIONAL PEARL

*A program to solve Sudoku*

RICHARD BIRD

*Programming Research Group, Oxford University*
*Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
email: bird@comlab.ox.ac.uk

There's no maths involved. You solve the puzzle with reasoning and logic.
*Advice on how to play Sudoku, The Independent Newspaper*

**1 Introduction**

The game of Sudoku is played on a 9 by 9 board. Given is a matrix of characters, such as

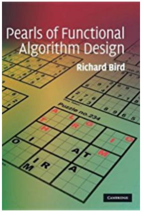
```
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

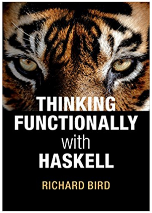
The idea is to fill in the dots with the digits 1 to 9 so that each row, each column and each of the component 3 by 3 boxes contains the digits 1 to 9 exactly once. In general there may be one, none or many solutions, though in a good Sudoku puzzle there is always a unique solution. Our aim in this pearl is to derive a Haskell program to solve Sudoku puzzles. Specifically, we will define a function

$$sudoku :: Board \to [Board]$$

for computing all the ways a given board may be filled. If we want only one solution we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

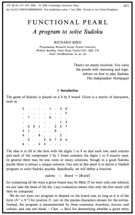
We do not want our program to depend on the board size, as long as it is of the form $(N^2 \times N^2)$ for positive $N$, nor on the precise characters chosen for the entries. Instead, the program is parameterized by three constants, *boardsize*, *boxsize* and *cellvals*, and one test $blank :: Char \to Bool$ for determining whether a given entry




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

# Welcome



JFP 16 (6): 671–679, 2006. © 2006 Cambridge University Press 671
doi:10.1017/S0956796806006058 First published online 7 July 2006 Printed in the United Kingdom

FUNCTIONAL PEARL

*A program to solve Sudoku*

RICHARD BIRD

*Programming Research Group, Oxford University*
*Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
*email:* bird@comlab.ox.ac.uk

There's no maths involved. You solve
the puzzle with reasoning and logic.
*Advice on how to play Sudoku,*
*The Independent Newspaper*

**1 Introduction**

The game of Sudoku is played on a 9 by 9 board. Given is a matrix of characters, such as

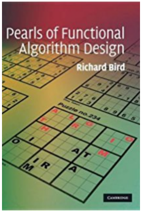
```
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

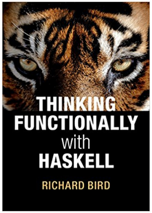
The idea is to fill in the dots with the digits 1 to 9 so that each row, each column and each of the component 3 by 3 boxes contains the digits 1 to 9 exactly once. In general there may be one, none or many solutions, though in a good Sudoku puzzle there is always a unique solution. Our aim in this pearl is to derive a Haskell program to solve Sudoku puzzles. Specifically, we will define a function

$$sudoku \;::\; Board \rightarrow [Board]$$

for computing all the ways a given board may be filled. If we want only one solution we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

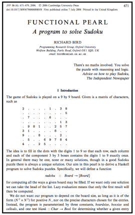
We do not want our program to depend on the board size, as long as it is of the form $(N^2 \times N^2)$ for positive $N$, nor on the precise characters chosen for the entries. Instead, the program is parameterized by three constants, *boardsize*, *boxsize* and *cellvals*, and one test *blank* :: *Char* → *Bool* for determining whether a given entry




- Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)
- Twelve short functions:
  - 5 easy ($\star$)
  - 4 medium ($\star\star$)
  - 3 challenging ($\star\star\star$)
- Programming techniques:
  - Top-down programming / wishful thinking
  - Wholemeal programming (prevents a disease called “indexitis”)

# Welcome



JFP 36 (6): 671–679, 2006. © 2006 Cambridge University Press 671
doi:10.1017/S0956796806006058 First published online 7 July 2006 Printed in the United Kingdom

FUNCTIONAL PEARL

*A program to solve Sudoku*

RICHARD BIRD

*Programming Research Group, Oxford University*
*Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
*email:* bird@comlab.ox.ac.uk

There's no maths involved. You solve
the puzzle with reasoning and logic.
*Advice on how to play Sudoku,*
*The Independent Newspaper*

**1 Introduction**

The game of Sudoku is played on a 9 by 9 board. Given is a matrix of characters, such as

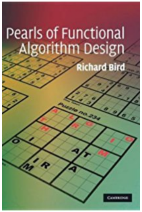
```
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

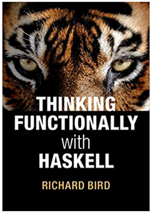
The idea is to fill in the dots with the digits 1 to 9 so that each row, each column and each of the component 3 by 3 boxes contains the digits 1 to 9 exactly once. In general there may be one, none or many solutions, though in a good Sudoku puzzle there is always a unique solution. Our aim in this pearl is to derive a Haskell program to solve Sudoku puzzles. Specifically, we will define a function

$$sudoku \;::\; Board \to [Board]$$

for computing all the ways a given board may be filled. If we want only one solution we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

We do not want our program to depend on the board size, as long as it is of the form $(N^2 \times N^2)$ for positive $N$, nor on the precise characters chosen for the entries. Instead, the program is parameterized by three constants, *boardsize*, *boxsize* and *cellvals*, and one test *blank* :: *Char* → *Bool* for determining whether a given entry




- Implement a Sudoku solver as described in (Bird, 2006, 2010, 2014)
- Twelve short functions:
  - 5 easy ($\star$)
  - 4 medium ($\star\star$)
  - 3 challenging ($\star\star\star$)
- Programming techniques:
  - Top-down programming / wishful thinking
  - Wholemeal programming (prevents a disease called “indexitis”)
  - Higher-order functions, recursion, point-free style

# How to play Sudoku

| | | | |
|---|---|---|---|
| | 2 | 4 | |
| 1 | | | 3 |
| 4 | | | 2 |
| | 1 | 3 | |

$N = 2$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 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 | |

$N = 3$

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 $N^2$.

# How to play Sudoku

| | | | |
|---|---|---|---|
| 3 | 2 | 4 | 1 |
| 1 | 4 | 2 | 3 |
| 4 | 3 | 1 | 2 |
| 2 | 1 | 3 | 4 |

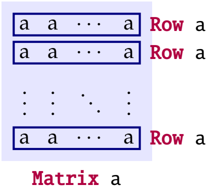
$N = 2$

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

$N = 3$

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

$$
\begin{array}{ll}
\boxed{a \quad a \quad \cdots \quad a} & \texttt{Row a} \\
\boxed{a \quad a \quad \cdots \quad a} & \texttt{Row a} \\
\vdots \quad \vdots \quad \ddots \quad \vdots & \\
\boxed{a \quad a \quad \cdots \quad a} & \texttt{Row a}
\end{array}
$$

`Matrix a`

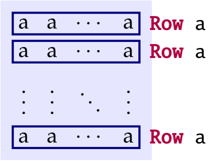
| | | | |
|---|---|---|---|
| 0 | 3 | 0 | 1 |
| 1 | 0 | 3 | 2 |
| 3 | 0 | 1 | 0 |
| 0 | 1 | 0 | 3 |

`Grid`

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

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

# Data types




Matrix a

| | | | |
|---|---|---|---|
| 0 | 3 | 0 | 1 |
| 1 | 0 | 3 | 2 |
| 3 | 0 | 1 | 0 |
| 0 | 1 | 0 | 3 |

Grid

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

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

We assume that digit zero indicates an empty cell:

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

# Exercise 1: `solve` [⋆]

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

# Exercise 1: `solve` [$\star$]

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

Given:

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

# Exercise 1: `solve` [$\star$]

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

Given:

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

```haskell
-- 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:

| | | | |
|---|---|---|---|
| 0 | 3 | 0 | 1 |
| 1 | 0 | 3 | 2 |
| 3 | 0 | 1 | 0 |
| 0 | 1 | 0 | 3 |

Grid

$\longrightarrow$

| | | | |
|---|---|---|---|
| 2 | 3 | 4 | 1 |
| 1 | 4 | 3 | 2 |
| 3 | 2 | 1 | 4 |
| 4 | 1 | 2 | 3 |

Grid

# Exercise 2: completions [$\star$]

```
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]
```

```haskell
-- 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:

| | | | |
|---|---|---|---|
| 0 | 3 | 0 | 1 |
| 1 | 0 | 3 | 2 |
| 3 | 0 | 1 | 0 |
| 0 | 1 | 0 | 3 |

**Grid**

$\longrightarrow$

| | | | |
|---|---|---|---|
| 1 | 3 | 1 | 1 |
| 1 | 1 | 3 | 2 |
| 3 | 1 | 1 | 1 |
| 1 | 1 | 1 | 3 |

**Grid**

| | | | |
|---|---|---|---|
| 1 | 3 | 1 | 1 |
| 1 | 1 | 3 | 2 |
| 3 | 1 | 1 | 1 |
| 1 | 1 | 2 | 3 |

**Grid**

...

| | | | |
|---|---|---|---|
| 4 | 3 | 4 | 1 |
| 1 | 4 | 3 | 2 |
| 3 | 4 | 1 | 4 |
| 4 | 1 | 4 | 3 |

**Grid**

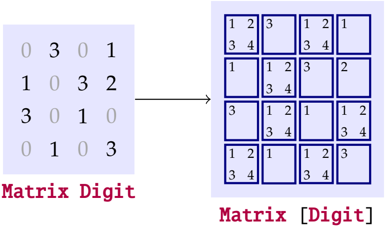
# Exercise 3: `choices` [$\star$]

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

# Exercise 3: `choices` [⋆]

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

Example:



# Exercise 3: `choices` [⋆]

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

Example:

| | | | |
|---|---|---|---|
| 0 | 3 | 0 | 1 |
| 1 | 0 | 3 | 2 |
| 3 | 0 | 1 | 0 |
| 0 | 1 | 0 | 3 |

`Matrix Digit`

$\longrightarrow$

| | | | |
|---|---|---|---|
| 1 2 3 4 | 3 | 1 2 3 4 | 1 |
| 1 | 1 2 3 4 | 3 | 2 |
| 3 | 1 2 3 4 | 1 | 1 2 3 4 |
| 1 2 3 4 | 1 | 1 2 3 4 | 3 |

`Matrix [Digit]`

Hint:

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

# Exercise 4: `expand` [⋆⋆]

```
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]]
```

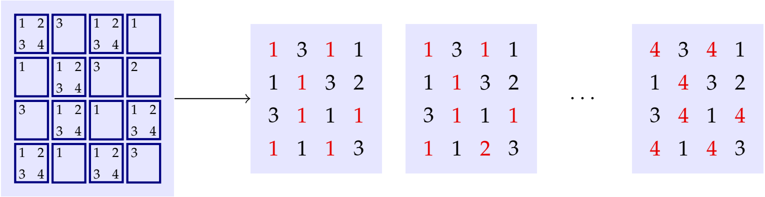
# Exercise 4: expand [⋆⋆]

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

Given:

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

Example:




**Matrix [Digit]**

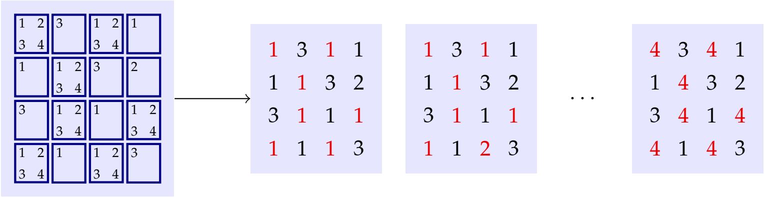
# Exercise 4: expand [⋆⋆]

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

Given:

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

Example:




`Matrix [Digit]`

Hints:

- First generate all combinations across each row
- Then combine those generated combinations

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

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

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

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

Example:

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

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

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

Example:

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

Hint:

- Use recursion:

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

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

Example:

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

Hint:

- Use recursion:

  xss

  $\boxed{1\ \ 2}\quad \boxed{3}\quad \boxed{4\ \ 5}$

# Exercise 5: cp [⋆⋆⋆]

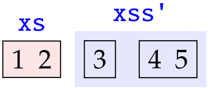
```haskell
cp :: [[a]] -> [[a]]
cp = undefined
```

Example:

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

Hint:

- Use recursion:



# Exercise 5: cp [⋆⋆⋆]

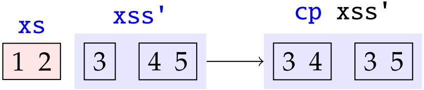
```
cp :: [[a]] -> [[a]]
cp = undefined
```

Example:

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

Hint:

- Use recursion:



# Exercise 5: cp [⋆⋆⋆]

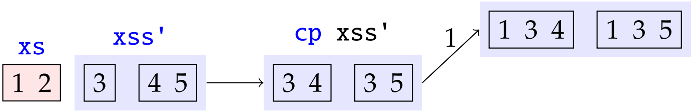
```haskell
cp :: [[a]] -> [[a]]
cp = undefined
```

Example:

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

Hint:

- Use recursion:



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

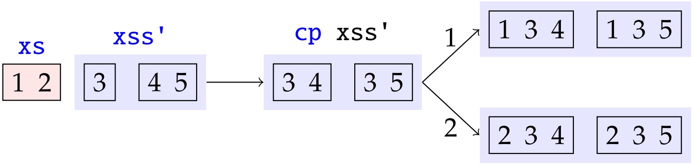
```haskell
cp :: [[a]] -> [[a]]
cp = undefined
```

Example:

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

Hint:

- Use recursion:



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

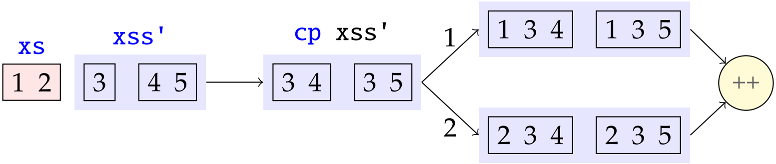
```
cp :: [[a]] -> [[a]]
cp = undefined
```

Example:

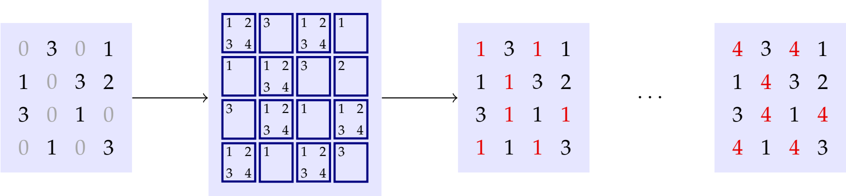
```
cp [[1, 2], [3, 4]] = [[1, 3], [1, 4], [2, 3], [2, 4]]
```

Hint:

- Use recursion:

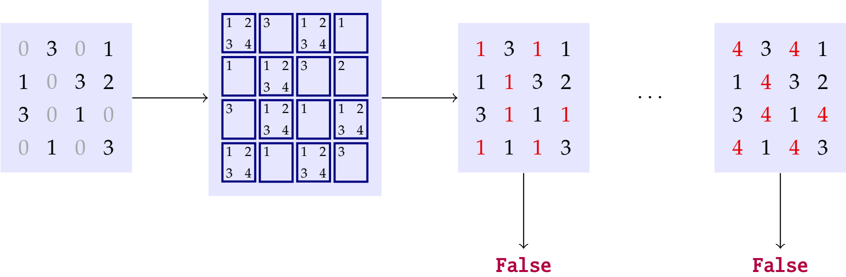


# Intermezzo




- Finished implementing the `completions` function

# Intermezzo




- Finished implementing the `completions` function
- Next, the `valid` function: test whether a grid is a valid solution

# Exercise 6: `valid` [$\star\star$]

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

# Exercise 6: `valid` [$\star\star$]

```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 [⋆⋆]

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

Given:

```
-- 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:

| | | | |
|---|---|---|---|
| 1 | 3 | 1 | 1 |
| 1 | 1 | 3 | 2 |
| 3 | 1 | 1 | 1 |
| 1 | 1 | 1 | 3 |

$\longrightarrow$ **False**

| | | | |
|---|---|---|---|
| 2 | 3 | 4 | 1 |
| 1 | 4 | 3 | 2 |
| 3 | 2 | 1 | 4 |
| 4 | 1 | 2 | 3 |

$\longrightarrow$ **True**

# Exercise 7: nodups [$\star\star$]

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

# Exercise 7: nodups [⋆⋆]

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

Examples:

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

# Exercise 7: nodups [$\star\star$]

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

Examples:

```haskell
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` [⋆]

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

Example:

$$\begin{matrix} 2 & 3 & 4 & 1 \\ 1 & 4 & 3 & 2 \\ 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 \end{matrix} \longrightarrow \begin{matrix} \boxed{2\quad 3\quad 4\quad 1} \\ \boxed{1\quad 4\quad 3\quad 2} \\ \boxed{3\quad 2\quad 1\quad 4} \\ \boxed{4\quad 1\quad 2\quad 3} \end{matrix}$$

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

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

# Exercise 9: `cols` [⋆⋆⋆]

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

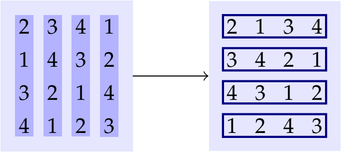
Example:

$$\begin{matrix} 2 & 3 & 4 & 1 \\ 1 & 4 & 3 & 2 \\ 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 \end{matrix} \longrightarrow \begin{matrix} \boxed{2 \quad 1 \quad 3 \quad 4} \\ \boxed{3 \quad 4 \quad 2 \quad 1} \\ \boxed{4 \quad 3 \quad 1 \quad 2} \\ \boxed{1 \quad 2 \quad 4 \quad 3} \end{matrix}$$

# Exercise 9: `cols` [⋆⋆⋆]

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

Example:




Hints:

- Use recursion
- Define a case for a one-row matrix; example:
  `cols [[1,2,3,4]] = [[1],[2],[3],[4]]`
- For the recursive case, use the `zipWith` function:
  `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

# Exercise 10: `boxs` [⋆⋆⋆]

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

# Exercise 10: boxs [⋆⋆⋆]

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

Given:

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

# Exercise 10: boxs [⋆⋆⋆]

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

Given:

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

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

# Exercise 10: boxs [⋆⋆⋆]

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

Given:

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

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

Example:

$$\begin{array}{cc|cc} 2 & 3 & 4 & 1 \\ 1 & 4 & 3 & 2 \\ \hline 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 \end{array} \longrightarrow \begin{array}{|cccc|} \hline 2 & 3 & 1 & 4 \\ \hline 4 & 1 & 3 & 2 \\ \hline 3 & 2 & 4 & 1 \\ \hline 1 & 4 & 2 & 3 \\ \hline \end{array}$$

# Exercise 10: boxs [⋆⋆⋆]

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

Given:

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

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

Example:

$$\begin{matrix} 2 & 3 & 4 & 1 \\ 1 & 4 & 3 & 2 \\ 3 & 2 & 1 & 4 \\ 4 & 1 & 2 & 3 \end{matrix} \longrightarrow \begin{matrix} \boxed{2 \quad 3 \quad 1 \quad 4} \\ \boxed{4 \quad 1 \quad 3 \quad 2} \\ \boxed{3 \quad 2 \quad 4 \quad 1} \\ \boxed{1 \quad 4 \quad 2 \quad 3} \end{matrix}$$

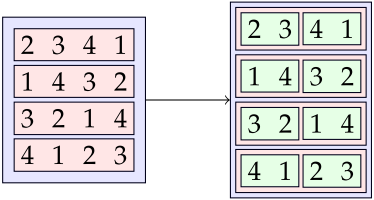
Hints:

- Use the previously defined `cols` function
- Chain five transformations (see next slide)

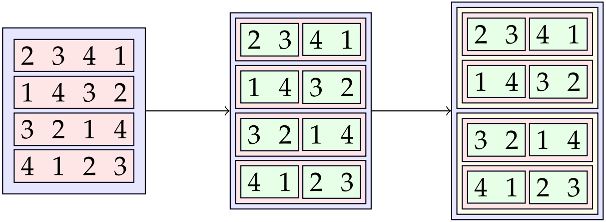
# Exercise 10: `boxs` [$\star\star\star$]

| | | | |
|---|---|---|---|
| 2 | 3 | 4 | 1 |
| 1 | 4 | 3 | 2 |
| 3 | 2 | 1 | 4 |
| 4 | 1 | 2 | 3 |

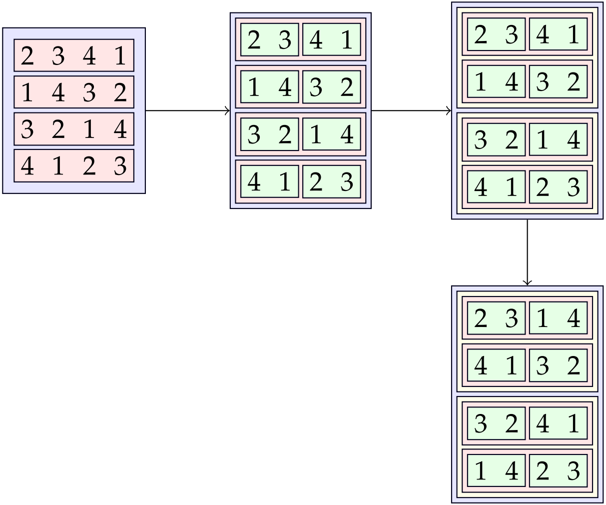
# Exercise 10: `boxs` [$\star\star\star$]



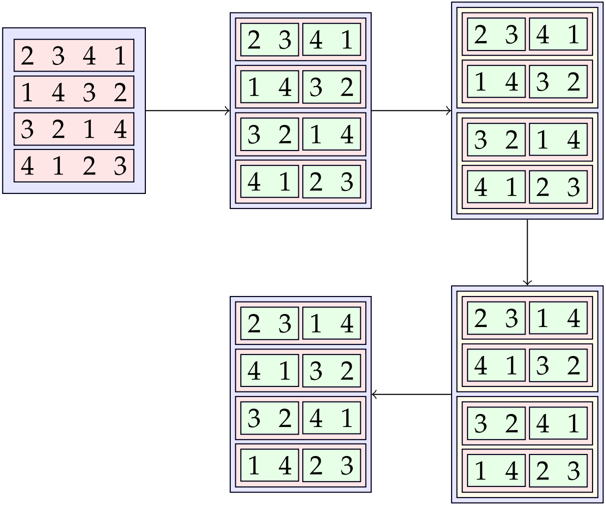
# Exercise 10: `boxs` [$\star\star\star$]



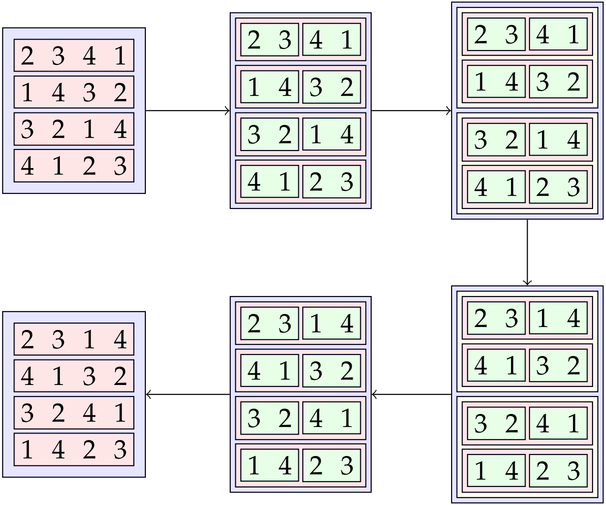
# Exercise 10: `boxs` [⋆ ⋆ ⋆]



# Exercise 10: boxs [⋆⋆⋆]



# Exercise 10: `boxs` [⋆⋆⋆]



# Exercise 11: `group` [$\star\star$]

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

# Exercise 11: `group` [$\star\star$]

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

Example:

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

# Exercise 12: ungroup [$\star$]

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

# Exercise 12: ungroup [$\star$]

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

Example:

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

# Exercise 12: `ungroup` [$\star$]

```
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

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct
- Equational reasoning to improve performance

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct
- Equational reasoning to improve performance
  - Define a function `prune` that eliminates early invalid solutions

```
filter valid . expand = filter valid . expand . prune
```

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct
- Equational reasoning to improve performance
  - Define a function `prune` that eliminates early invalid solutions

    ```
    filter valid . expand = filter valid . expand . prune
    ```
  - It is not hard to define a function to prune a row (exercise):

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

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct
- Equational reasoning to improve performance
  - Define a function `prune` that eliminates early invalid solutions

    ```
    filter valid . expand = filter valid . expand . prune
    ```
  - It is not hard to define a function to prune a row (exercise):

    ```
    pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]
    ```
  - Equational reasoning to define `prune` in terms of `pruneRow`

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct
- Equational reasoning to improve performance
  - Define a function `prune` that eliminates early invalid solutions

    ```
    filter valid . expand = filter valid . expand . prune
    ```
  - It is not hard to define a function to prune a row (exercise):

    ```
    pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]
    ```
  - Equational reasoning to define `prune` in terms of `pruneRow`
    - The function `pruneRow` satisfies the equation

      ```
      filter nodups . cp = filter nodups . cp . pruneRow
      ```

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct
- Equational reasoning to improve performance
  - Define a function `prune` that eliminates early invalid solutions

    `filter valid . expand = filter valid . expand . prune`
  - It is not hard to define a function to prune a row (exercise):

    `pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]`
  - Equational reasoning to define `prune` in terms of `pruneRow`
    - The function `pruneRow` satisfies the equation

      `filter nodups . cp = filter nodups . cp . pruneRow`
    - Expand the expression `filter valid . expand`

# That's all folks

- Time to solve some Sudokus
- The current approach is inefficient, but correct
- Equational reasoning to improve performance
  - Define a function `prune` that eliminates early invalid solutions

    `filter valid . expand = filter valid . expand . prune`
  - It is not hard to define a function to prune a row (exercise):

    `pruneRow [[4],[1,2],[1],[1,3]] = [[4],[2],[1],[3]]`
  - Equational reasoning to define `prune` in terms of `pruneRow`
    - The function `pruneRow` satisfies the equation

      `filter nodups . cp = filter nodups . cp . pruneRow`
    - Expand the expression `filter valid . expand`
    - Use the above equation and compress back the formula

# Further references

- Richard Bird's papers and books (Bird, 2006, 2010, 2014)
- Conor McBride's Sudoku solver using applicative and traversable: https://stackoverflow.com/a/10242673/474311

# 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.