

N-Queens Combinatorial Problem

Polyglot **FP** for **F**un and **P**rofit – Haskell and **Scala** 

first see the problem solved using the **List monad** and a **Scala for comprehension**

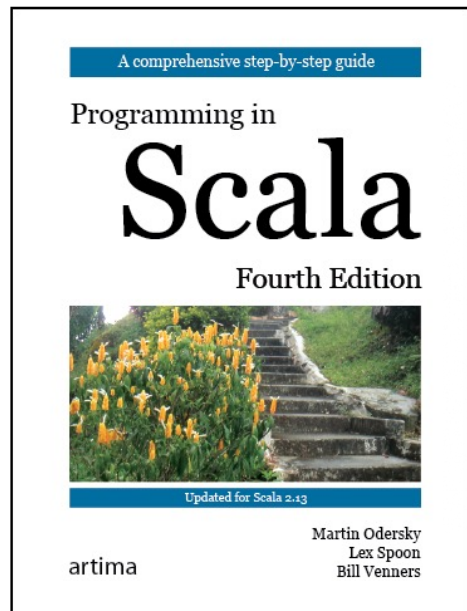
then see the **Scala** program translated into **Haskell**, both using a **do expressions** and using a **List comprehension**

understand how the **Scala for comprehension** is **desugared**, and what role the **withFilter** function plays

also understand how the **Haskell do expressions** and **List comprehension** are **desugared**, and what role the **guard** function plays

Part 1

based on, and inspired by, the work of



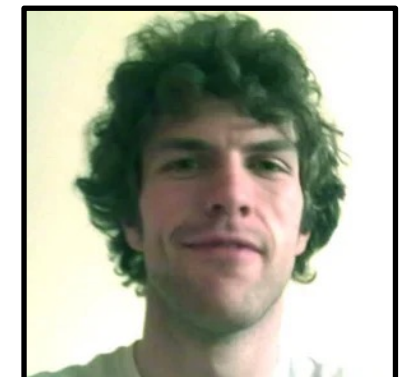
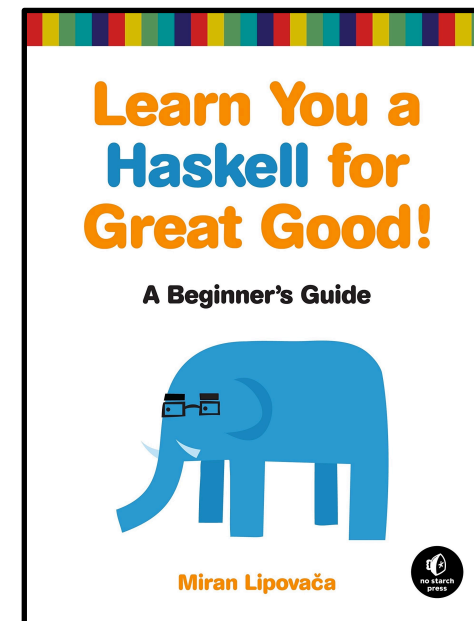
Martin Odersky



Lex Spoon



Bill Venners



Miran Lipovača

slides by



 @philip_schwarz

 slideshare

<https://www.slideshare.net/pjschwarz>



 @philip_schwarz

Yes, I know, **Lex Spoon** looks a bit too young in that photo, and **Bill Venner's** hair doesn't look realistic in that drawing. **Martin Odersky's** drawing is not bad at all.

By the way, don't be put off by the fact that the book image below is of the fourth edition. Excerpts from the **N-queens** section of that edition are great, and by the way, in the fifth edition (new for **Scala 3**) I don't see that section (it may have moved to upcoming book **Advanced Programming in Scala**). Also, I do use **Scala 3** in this slide deck.



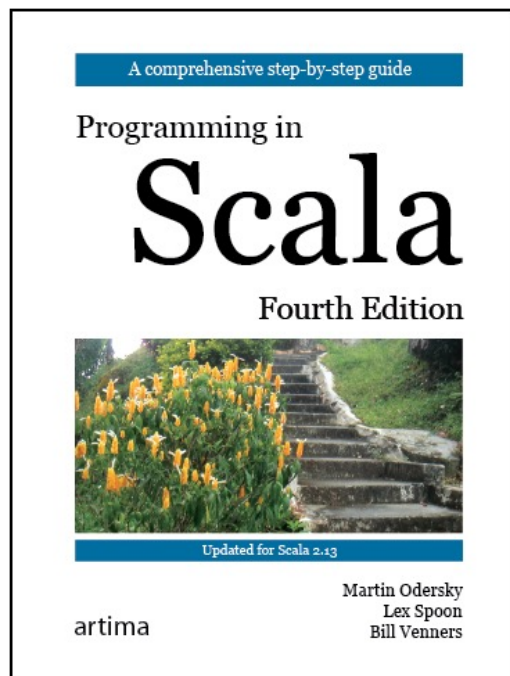
Lex Spoon



Martin Odersky



Bill Venners





A particularly suitable application area of **for expressions** are **combinatorial puzzles**.

An example of such a puzzle is the **8-queens problem**: Given a standard chess-board, place eight queens such that no queen is **in check** from any other (a queen can **check** another piece if they are on the same column, row, or diagonal).

To find a solution to this problem, it's actually simpler to generalize it to chess-boards of arbitrary size.

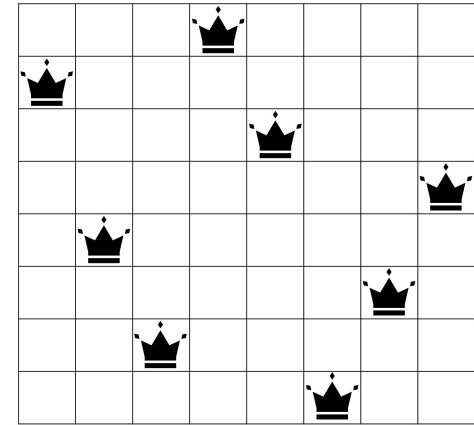
Hence the problem is to place N **queens** on a chess-board of $N \times N$ squares, where the size N is arbitrary.

We'll start numbering cells at one, so the upper-left cell of an $N \times N$ board has coordinate $(1,1)$ and the lower-right cell has coordinate (N,N) .

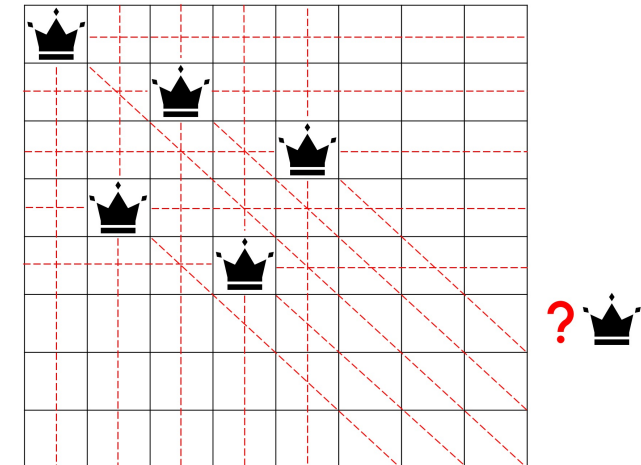
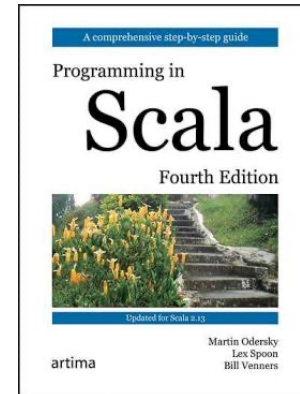
To solve the **N-queens problem**, note that **you need to place a queen in each row**. So you could place queens in successive rows, each time **checking that a newly placed queen is not in check from any other queens that have already been placed**.

In the course of this search, **it might happen that a queen that needs to be placed in row k would be in check in all fields of that row from queens in row 1 to $k - 1$** . In that case, you need to abort that part of the search in order to continue with a different configuration of queens in columns 1 to $k - 1$.

Here on the right is a sample solution to the puzzle.



And below is an example in which there is nowhere to place queen number 6 because every other cell on the board is **in check** from queens 1 to 5.



An **imperative solution** to this problem would place queens one by one, moving them around on the board.

But it looks difficult to come up with a scheme that really tries all possibilities.

A **more functional approach** represents a solution directly, as a value. A solution consists of a list of coordinates, one for each queen placed on the board.

Note, however, that **a full solution can not be found in a single step. It needs to be built up gradually, by occupying successive rows with queens.**

This suggests a **recursive algorithm**.

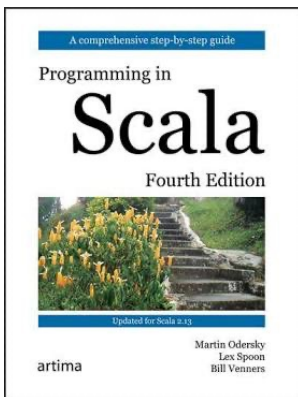
Assume you have already generated all solutions of placing k queens on a board of size $N \times N$, where k is less than N .

Each solution can be represented by a list of length k of coordinates (row, column), where both row and column numbers range from 1 to N .

It's convenient to treat these partial solution lists as **stacks**, where the coordinates of the queen in row k come first in the list, followed by the coordinates of the queen in row $k - 1$, and so on.

The bottom of the **stack** is the coordinate of the queen placed in the first row of the board.

All solutions together are represented as a **list of lists**, with one element for each solution.



R C
O O
W L

			♙					(1, 4)
♙								(2, 1)
				♙				(3, 5)
							♙	(4, 8)
	♙							(5, 2)
						♙		(6, 7)
		♙						(7, 3)
				♙				(8, 6)

See below for the list representing the above solution to the **8-queens** puzzle.



List((8, 6), (7, 3),
(6, 7), (5, 2),
(4, 8), (3, 5),
(2, 1), (1, 4))

Now, to place the next queen in row $k + 1$, generate all possible extensions of each previous solution by one more queen.

This yields another list of solutions lists, this time of length $k + 1$.

Continue the process until you have obtained all solutions of the size of the chess-board N .

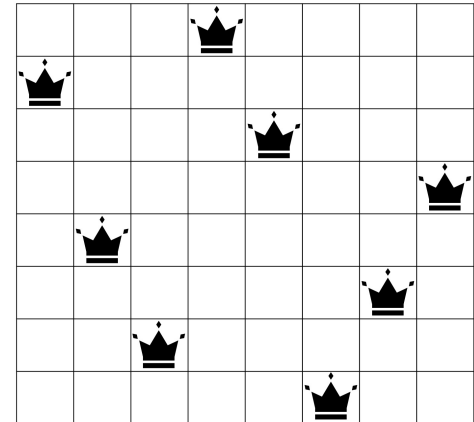
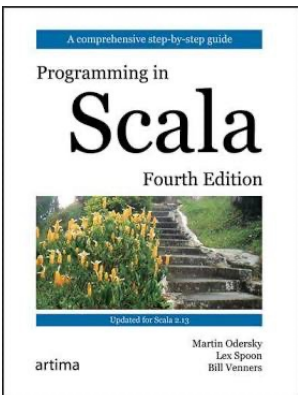
This **algorithmic idea** is embodied in function **placeQueens** below:

```
def queens(n: Int): List[List[(Int,Int)]] = {  
  def placeQueens(k: Int): List[List[(Int,Int)]] =  
    if (k == 0)  
      List(List())  
    else  
      for {  
        queens <- placeQueens(k - 1)  
        column <- 1 to n  
        queen = (k, column)  
        if isSafe(queen, queens)  
      } yield queen :: queens  
  placeQueens(n)  
}
```

The outer function **queens** in the program above simply calls **placeQueens** with the size of the board n as its argument.

The task of the function application **placeQueens**(k) is to generate all partial solutions of length k in a list.

Every element of the list is one solution, represented by a list of length k . So **placeQueens** returns a **list of lists**.



```
List((8, 6), (7, 3),  
      (6, 7), (5, 2),  
      (4, 8), (3, 5),  
      (2, 1), (1, 4))
```

If the parameter k to `placeQueens` is 0 , this means that it needs to generate all solutions of placing zero queens on zero rows. There is only one solution: place no queen at all. This solution is represented by the empty list. So if k is zero, `placeQueens` returns `List(List())`, a list consisting of a single element that is the empty list.

Note that this is quite different from the empty list `List()`. If `placeQueens` returns `List()`, this means no solutions, instead of a single solution consisting of no placed queens.

In the other case, where k is not zero, all the work of `placeQueens` is done in the **for expression**.

The **first generator** of that **for expression** iterates through all solutions of placing $k - 1$ queens on the board.

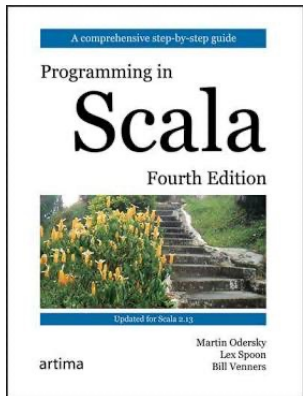
The **second generator** iterates through all possible columns on which the k^{th} queen might be placed.

The **third part** of the **for expression** defines the newly considered queen position to be the pair consisting of row k and each produced column.

The **fourth part** of the **for expression** is a filter which checks with `isSafe` whether the new queen is safe from check by all previous queens (the definition of `isSafe` will be discussed a bit later.)

If the new queen is not **in check** from any other queens, it can form part of a partial solution, so `placeQueens` generates with `queen :: queens` a new solution.

If the new queen is not safe from **check**, the filter returns **false**, so no solution is generated.



Here is the **for expression** again, for reference.



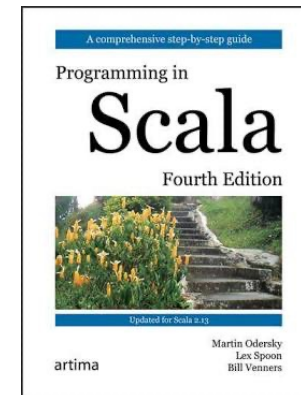
```
for {  
  queens <- placeQueens(k - 1)  
  column <- 1 to n  
  queen = (k, column)  
  if isSafe(queen, queens)  
} yield queen :: queens
```

The only remaining bit is the `isSafe` method, which is used to check whether a given queen is **in check** from any other element in a list of queens.

Here is the definition:

```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 || // same row
  q1._2 == q2._2 || // same column
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```



The `isSafe` method expresses that a queen is safe with respect to some other queens if it is not **in check** from any other queen.

The `inCheck` method expresses that queens `q1` and `q2` are mutually **in check**.

It returns true in one of three cases:

1. If the two queens have the **same row coordinate**.
2. If the two queens have the **same column coordinate**.
3. If the two queens are on **the same diagonal** (*i.e.*, the difference between their rows and the difference between their columns are the same).

The first case – that the two queens have the same row coordinate – cannot happen in the application because `placeQueens` already takes care to place each queen in a different row. So you could remove the test without changing the functionality of the program.





6.3 Combinatorial Search Example

830 views • 11 Sept 2017



Algorithm

We can solve this problem with a recursive algorithm:

- ▶ Suppose that we have already generated all the solutions consisting of placing $k-1$ queens on a board of size n .
- ▶ Each solution is represented by a list (of length $k-1$) containing the numbers of columns (between 0 and $n-1$).
- ▶ The column number of the queen in the $k-1$ th row comes first in the list, followed by the column number of the queen in row $k-2$, etc.
- ▶ The solution set is thus represented as a set of lists, with one element for each solution.
- ▶ Now, to place the k th queen, we generate all possible extensions of each solution preceded by a new queen:

Example: N-Queens

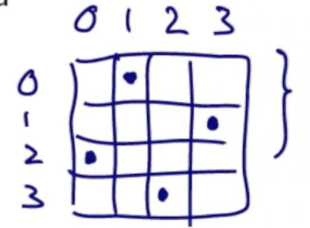
The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.

- ▶ In other words, there can't be two queens in the same row, column, or diagonal.

We now develop a solution for a chessboard of any size, not just 8.

One way to solve the problem is to place a queen on each row.

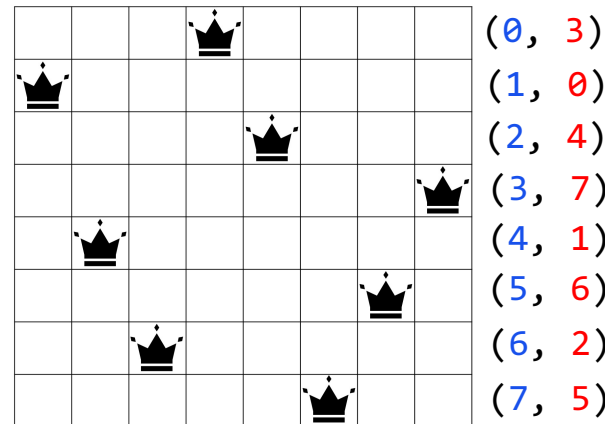
Once we have placed $k - 1$ queens, one must place the k th queen in a column where it's not "in check" with any other queen on the board.



List(0, 3, 1)



List(2, 0, 3, 1)



List(5, 2, 6, 1, 7, 4, 0, 3)

In online course [FP in Scala](#), the board coordinates are **zero-based**, and since it is obvious that the n^{th} queen is in row n , a solution is a list of **column indices** rather than a list of **coordinate pairs**.



@philip_schwarz



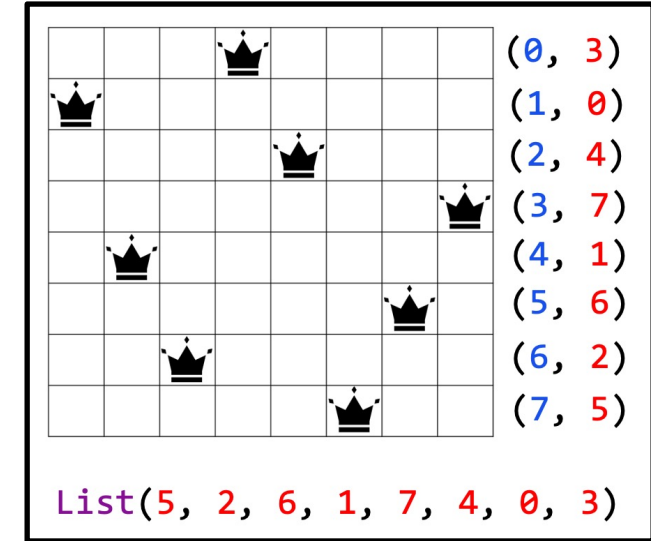
```
object nqueens {
  def queens(n: Int): Set[List[Int]] = {
    def placeQueens(k: Int): Set[List[Int]] =
      if (k == 0) Set(List())
      else
        for {
          queens <- placeQueens(k - 1)
          col <- 0 until n
          if isSafe(col, queens)
        } yield col :: queens
    placeQueens(n)
  }
}
```



> queens: (n: Int)Set[List[Int]]

```
def isSafe(col: Int, queens: List[Int]): Boolean = {
  val row = queens.length
  val queensWithRow = (row - 1 to 0 by -1) zip queens
  queensWithRow forall {
    case (r, c) => col != c && math.abs(col - c) != row - r
  }
}
```

> isSafe: (col: Int, queens: List[Int])Bo

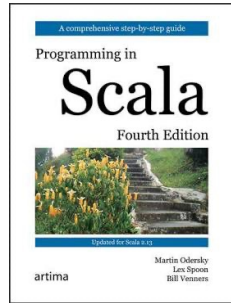


Because a column index is sufficient to indicate the coordinates of the n^{th} queen, we can see that the type of `isSafe`'s first parameter is now `Int`, i.e. a column index, rather than `(Int, Int)` i.e. a coordinate pair.

Also, the type of a solution is `List[Int]` rather than `List[(Int, Int)]`, and solutions are returned in a `Set`, rather than a `List`, to distinguish between a single solution, which is an ordered list, and the collection of all solutions, which is not ordered (though it could be, if we so wanted).



Here are the two versions of the program side by side.



Functional Programming Principles in Scala

1.18K subscribers

```
def queens(n: Int): List[List[(Int, Int)]] = {
  def placeQueens(k: Int): List[List[(Int, Int)]] =
    if (k == 0)
      List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
      } yield queen :: queens
  placeQueens(n)
}
```



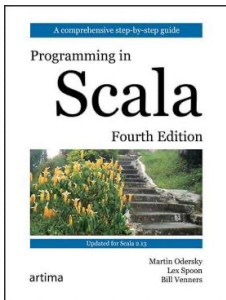
```
def queens(n: Int): Set[List[Int]] = {
  def placeQueens(k: Int): Set[List[Int]] =
    if (k == 0)
      Set(List())
    else
      for {
        queens <- placeQueens(k - 1)
        col <- 0 until n
        if isSafe(col, queens)
      } yield col :: queens
  placeQueens(n)
}
```



```
def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
  q1._1 == q2._1 || // same row
  q1._2 == q2._2 || // same column
  (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```

```
def isSafe(col: Int, queens: List[Int]): Boolean = {
  val row = queens.length
  val queensWithRow = (row - 1 to 0 by -1) zip queens
  queensWithRow forall
    { case (r, c) => col != c && math.abs(col - c) != row - r }
}
```



Same as on the previous slide, but with really minor tweaks thanks to [Scala 3](#)



Functional Programming Principles in Scala

1.18K subscribers

```
def queens(n: Int): List[List[(Int, Int)]] =
  def placeQueens(k: Int): List[List[(Int, Int)]] =
    if k == 0
    then List(List())
    else
      for
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
      yield queen :: queens
    placeQueens(n)

def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(queen1: (Int, Int), queen2: (Int, Int)) =
  queen1(0) == queen2(0) || // same row
  queen1(1) == queen2(1) || // same column
  (queen1(0) - queen2(0)).abs == (queen1(1) - queen2(1)).abs // on diagonal
```



```
def queens(n: Int): Set[List[Int]] =
  def placeQueens(k: Int): Set[List[Int]] =
    if k == 0
    then Set(List())
    else
      for
        queens <- placeQueens(k - 1)
        col <- 0 until n
        if isSafe(col, queens)
      yield col :: queens
    placeQueens(n)

def isSafe(col: Int, queens: List[Int]): Boolean =
  val row = queens.length
  val queensWithRow = (row - 1 to 0 by -1) zip queens
  queensWithRow forall
    { case (r, c) => col != c && math.abs(col - c) != row - r }
```



In the next part of this slide deck, we are going to be using the program on the right.

The `queens` and `placeQueens` functions are pretty much the same as in the second version on the previous slide, but board coordinates are one-based, and the two functions return a `List` rather than a `Set`.

The logic for determining if a queen is safe from previously placed queens, is organised in a way that is in part a hybrid between the two programs on the previous slide.



 @philip_schwarz

```
def queens(n: Int): List[List[Int]] =  
  def placeQueens(k: Int): List[List[Int]] =  
    if k == 0  
    then List(List())  
    else  
      for  
        queens <- placeQueens(k - 1)  
        queen <- 1 to n  
        if safe(queen, queens)  
      yield queen :: queens  
  placeQueens(n)
```

			♔					(1, 4)
♔								(2, 1)
			♔					(3, 5)
							♔	(4, 8)
	♔							(5, 2)
						♔		(6, 7)
		♔						(7, 3)
					♔			(8, 6)

List(6, 3, 7, 2, 8, 5, 1, 4)

```
def safe(queen: Int, queens: List[Int]): Boolean =  
  val (row, column) = (queens.length, queen)  
  val safe: ((Int,Int)) => Boolean = (nextRow, nextColumn) =>  
    column != nextColumn && !onDiagonal(column, row, nextColumn, nextRow)  
  zipWithRows(queens) forall safe
```

```
def onDiagonal(row: Int, column: Int, otherRow: Int, otherColumn: Int) =  
  math.abs(row - otherRow) == math.abs(column - otherColumn)
```

```
def zipWithRows(queens: List[Int]): Iterable[(Int,Int)] =  
  val rowCount = queens.length  
  val rowNumbers = rowCount - 1 to 0 by -1  
  rowNumbers zip queens
```


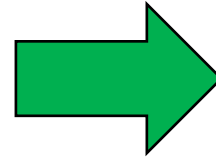
```
assert(queens(4) == List(List(3, 1, 4, 2), List(2, 4, 1, 3)))  
assert(queens(6) == List(List(3, 6, 2, 5, 1, 4),  
                          List(4, 1, 5, 2, 6, 3),  
                          List(5, 3, 1, 6, 4, 2),  
                          List(2, 4, 6, 1, 3, 5)))
```




The first thing we are going to do, is translate our **N-queens** program from **Scala** into **Haskell**.

Let's start with the **queens** and **placeQueens** functions.

```
def queens(n: Int): List[List[Int]] =
  def placeQueens(k: Int): List[List[Int]] =
    if k == 0
    then List(List())
    else
      for
        queens <- placeQueens(k - 1)
        queen <- 1 to n
        if safe(queen, queens)
      yield queen :: queens
  placeQueens(n)
```

```
queens n = placeQueens n
  where
    placeQueens 0 = [[]]
    placeQueens k =
      do
        queens <- placeQueens(k-1)
        queen <- [1..n]
        guard (safe queen queens)
        return (queen:queens)
```



In **Scala** we use a **for comprehension**, whereas in **Haskell** we use a **do expression**.

In **Scala**, the filtering is done by

```
if safe(queen, queens)
```

whereas in **Haskell** it is done by

```
guard (safe queen queens)
```

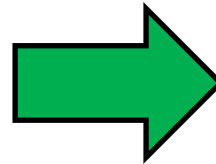


If you are asking yourself what the **guard** function is, don't worry: we'll be looking at it soon.



We can simplify the **Haskell** version a little bit by eliminating the calls to **guard** and **return**, which can be achieved by switching from a **do expression** to a **list comprehension**.

```
queens n = placeQueens n
  where
    placeQueens 0 = [[]]
    placeQueens k = do
      queens <- placeQueens(k-1)
      queen <- [1..n]
      guard (safe queen queens)
      return (queen:queens)
```



```
queens n = placeQueens n
  where
    placeQueens 0 = [[]]
    placeQueens k =
      [queen:queens |
       queens <- placeQueens(k-1),
       queen <- [1..n],
       safe queen queens]
```



If you are asking yourself how the **list comprehension** eliminates the need for **guard** and **return**, don't worry: we'll be looking at that soon.



And here is the translation
of the whole program



```
def queens(n: Int): List[List[Int]] =  
  def placeQueens(k: Int): List[List[Int]] =  
    if k == 0  
    then List(List())  
    else  
      for  
        queens <- placeQueens(k - 1)  
        queen <- 1 to n  
        if safe(queen, queens)  
      yield queen :: queens  
  placeQueens(n)
```

```
def safe(queen: Int, queens: List[Int]): Boolean =  
  val (row, column) = (queens.length, queen)  
  val safe: ((Int,Int)) => Boolean = (nextRow, nextColumn) =>  
    column != nextColumn && !onDiagonal(column, row, nextColumn, nextRow)  
  zipWithRows(queens) forall safe
```

```
def onDiagonal(row: Int, column: Int, otherRow: Int, otherColumn: Int) =  
  math.abs(row - otherRow) == math.abs(column - otherColumn)
```

```
def zipWithRows(queens: List[Int]): Iterable[(Int,Int)] =  
  val rowCount = queens.length  
  val rowNumbers = rowCount - 1 to 0 by -1  
  rowNumbers zip queens
```

```
queens n = placeQueens n  
  where  
    placeQueens 0 = [[]]  
    placeQueens k = [queen:queens |  
      queens <- placeQueens(k-1),  
      queen <- [1..n],  
      safe queen queens]
```

```
safe queen queens = all safe (zipWithRows queens)  
  where  
    safe (r,c) = c /= col && not (onDiagonal col row c r)  
    row = length queens  
    col = queen
```

```
onDiagonal row column otherRow otherColumn =  
  abs (row - otherRow) == abs (column - otherColumn)
```

```
zipWithRows queens = zip rowNumbers queens  
  where  
    rowCount = length queens  
    rowNumbers = [rowCount-1,rowCount-2..0]
```



Let's try out both the **Scala** program and the **Haskell** one.

 @philip_schwarz

```
scala> queens(4)
val res0: List[List[Int]] = List(List(3, 1, 4, 2), List(2, 4, 1, 3))

scala> queens(5)
val res1: List[List[Int]] = List(List(4, 2, 5, 3, 1), List(3, 5, 2, 4, 1), List(5, 3, 1, 4, 2), List(4, 1, 3, 5, 2),
                                List(5, 2, 4, 1, 3), List(1, 4, 2, 5, 3), List(2, 5, 3, 1, 4), List(1, 3, 5, 2, 4),
                                List(3, 1, 4, 2, 5), List(2, 4, 1, 3, 5))

scala> queens(6)
val res2: List[List[Int]] = List(List(5, 3, 1, 6, 4, 2), List(4, 1, 5, 2, 6, 3),
                                List(3, 6, 2, 5, 1, 4), List(2, 4, 6, 1, 3, 5))
```



```
haskell> queens 4
[[3,1,4,2],[2,4,1,3]]

haskell> queens 5
[[4,2,5,3,1],[3,5,2,4,1],[5,3,1,4,2],[4,1,3,5,2],
 [5,2,4,1,3],[1,4,2,5,3],[2,5,3,1,4],[1,3,5,2,4],
 [3,1,4,2,5],[2,4,1,3,5]]

haskell> queens 6
[[5,3,1,6,4,2],[4,1,5,2,6,3],
 [3,6,2,5,1,4],[2,4,6,1,3,5]]
```





For $N > 6$, the number of solutions is too large for the solutions to be displayed simply by printing them to the screen.

Let's at least check that we get the expected number of solutions.

<i>N</i>	N-queens solution count
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2,680
12	14,200
13	73,712
14	365,596

15	2,279,184
16	14,772,512
17	95,815,104
18	666,090,624
19	4,968,057,848
20	39,029,188,884
21	314,666,222,712
22	2,691,008,701,644
23	24,233,937,684,440
24	227,514,171,973,736
25	2,207,893,435,808,352
26	22,317,699,616,364,044
27	234,907,967,154,122,528

```
scala> queens(8).size
val res0: Int = 92

scala> queens(9).size
val res1: Int = 352

scala> queens(10).size
val res2: Int = 724

scala> queens(11).size
val res3: Int = 2680

scala> queens(12).size
val res4: Int = 14200

scala> queens(13).size
val res5: Int = 73712

scala> queens(14).size
val res6: Int = 365596
```

```
Haskell> length (queens 8)
92

Haskell> length (queens 9)
352

Haskell> length (queens 10)
724

Haskell> length (queens 11)
2680

Haskell> length (queens 12)
14200

Haskell> length (queens 13)
73712

Haskell> length (queens 14)
365596
```



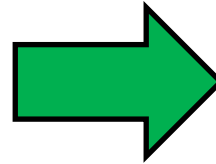
 @philip_schwarz

In the next five slides, we are going to look at how the **Scala** **for comprehension** is implemented under the hood.



Here is how the **N-queens for comprehension**, without the filter, is implemented, i.e. what it looks like when it is **desugared** (when its syntactic sugar is removed).

```
for
  queens <- placeQueens(k - 1)
  queen <- 1 to n
yield queen :: queens
```

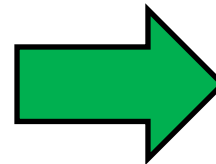


```
placeQueens(k - 1) flatMap { queens =>
  (1 to n) map { queen =>
    queen :: queens
  }
}
```



To aid our understanding, here is a similar **for comprehension** that is simpler to understand, but can be run at the **REPL**, together with its **desugared** version.

```
for
  nums <- List(List(1,2),List(3,4))
  num <- List(10,20)
yield num :: nums
```



```
List(List(1,2),List(3,4)) flatMap { nums =>
  List(10,20) map { num =>
    num :: nums
  }
}
```



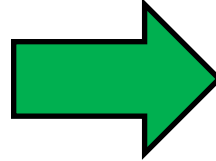
And here is the value produced by both the **for comprehension** and its **desugared** equivalent.

```
List(List(10, 1, 2), List(20, 1, 2), List(10, 3, 4), List(20, 3, 4))
```



Same as on the previous slide, except that now we are adding a filter to the **for comprehension**.
When the filter is **desugared**, it results in a call to **List**'s **withFilter** function.

```
for
  queens <- placeQueens(k - 1)
  queen <- 1 to n
  if safe(queen, queens)
yield queen :: queens
```

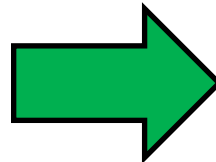


```
placeQueens(k - 1) flatMap { queens =>
  (1 to n) withFilter { queen =>
    safe(queen, queens)
  } map { queen =>
    queen :: queens
  }
}
```



To aid our understanding, here is a similar **for comprehension** that is simpler to understand, and that can be run at the **REPL**, together with its **desugared** version.

```
for
  nums <- List(List(1,2),List(3,4))
  num <- List(10,20)
  if (num > 15)
yield num :: nums
```



```
List(List(1,2),List(3,4)) flatMap { nums =>
  List(10,20) withFilter { num =>
    num > 15
  } map { num =>
    num :: nums
  }
}
```



And here is the value produced by both the **for comprehension** and its **desugared** equivalent.

List(List(20, 1, 2), List(20, 3, 4))



 @philip_schwarz

Let's see what the **withFilter** function does:

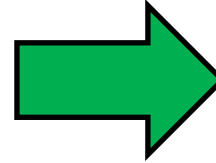
- it is applied to a **monad**, in this case a **List**
- it takes a **predicate**, i.e. a function that takes a parameter and returns either **true** or **false**.
- it applies the **predicate** to each value yielded by the **monad** (in this case, each value contained in a **List**)
- it filters out the values for which the **predicate** returns **false**
- It lets through the values for which the **predicate** returns **true**

On the next slide, we look at an example of how the behaviour of a **for comprehension** (or its **desugared** equivalent) is affected by the behaviour of the **withFilter** function.

withFilter
lets through
some elements

```
for
  nums <- List(List(1,2),List(3,4))
  num <- List(10,20)
  if (num > 15)
yield num :: nums
```

List(List(20, 1, 2), List(20, 3, 4))

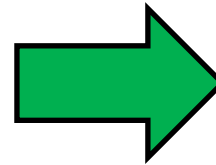


```
List(List(1,2),List(3,4)) flatMap { nums =>
  List(10,20) withFilter { num =>
    num > 15
  } map { num =>
    num :: nums
  }
}
```

withFilter
lets through
all elements

```
for
  nums <- List(List(1,2),List(3,4))
  num <- List(10,20)
  if (true)
yield num :: nums
```

List(List(10, 1, 2), List(20, 1, 2), List(10, 3, 4), List(20, 3, 4))

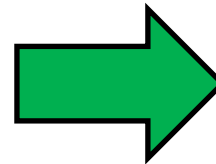


```
List(List(1,2),List(3,4)) flatMap { nums =>
  List(10,20) withFilter { num =>
    true
  } map { num =>
    num :: nums
  }
}
```

withFilter
lets through
no elements

```
for
  nums <- List(List(1,2),List(3,4))
  num <- List(10,20)
  if (false)
yield num :: nums
```

List()



```
List(List(1,2),List(3,4)) flatMap { nums =>
  List(10,20) withFilter { num =>
    false
  } map { num =>
    num :: nums
  }
}
```

Notice that the **predicate** function passed to **withFilter** may or may not use its parameter. Like the **map** and **flatMap** functions, **withFilter** is used to bind the value(s) yielded by a **monad** (if any) to a variable name which is then available for use in subsequent computations.



The difference is that while the functions passed to **map** and **flatMap** are almost certain to use the variable, in the case of **withFilter**, it may choose to influence the overall computation without actually using the variable.



Before we move on, here is a final, illustration of what the `withFilter` function does.

```
scala> List(1,2,3).withFilter(n => true).map(identity)
val res0: List[Int] = List(1, 2, 3)

scala> List(1,2,3).withFilter(n => n > 1).map(identity)
val res1: List[Int] = List(2, 3)

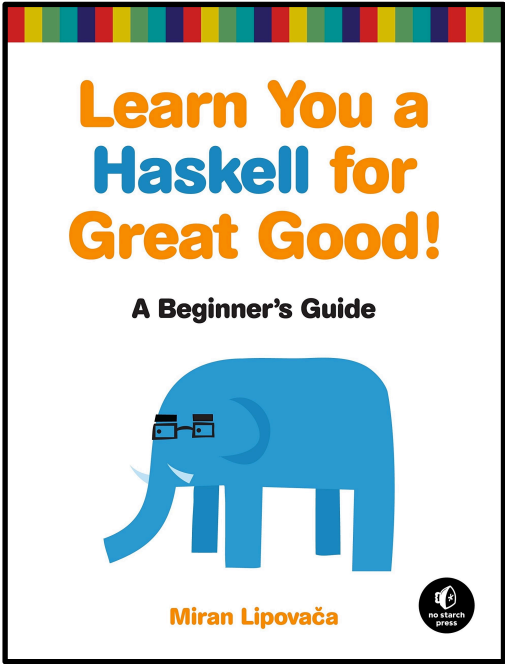
scala> List(1,2,3).withFilter(n => false).map(identity)
val res2: List[Int] = List()
```



Earlier we saw that the **do expression** that is the **Haskell** translation of the **Scala for comprehension**, involves the use of the **guard** function.

Also, later we intend to look at how the **Haskell do expression** is implemented under the hood, which involves both the **guard** function and the **(>>)** function.

So, in the next three slides, we are going to look at how **Miran Lipovača** explains **guard** and **(>>)**.



Here is the default implementation of the (`>>`) function:

```
(>>) :: (Monad m) => ma -> mb -> mb
m >> n = m >>= \_ -> n
```

Normally, passing some value to a function that ignores its parameter and always returns some predetermined value always results in that predetermined value. With **monads** however, their context and meaning must be considered as well.

Here is how `>>` acts with **Maybe**:

```
ghci> Nothing >> Just 3
Nothing

ghci> Just 3 >> Just 4
Just 4

ghci> Just 3 >> Nothing
Nothing
```

Here on the right is how `>>` acts with lists.

```
ghci> [] >> [3]
[]

ghci> [3] >> [4]
[4]

ghci> [3] >> []
[]
```

`>>=` is Haskell's equivalent of Scala's `flatMap` and `_ -> n` is Haskell's equivalent of Scala's anonymous function `_ => n`.



Miran Lipovača

Same as above, but using the **Scala Cats** library:

```
import cats._
import cats.implicits._
```



```
scala> None >> 3.some
val res0: Option[Int] = None

scala> 3.some >> 4.some
val res1: Option[Int] = Some(4)

scala> 3.some >> None
val res2: Option[Nothing] = None

scala> None >> None
val res3: Option[Nothing] = None
```

```
scala> List() >> List(3)
val res0: List[Int] = List()

scala> List(3) >> List(4)
val res1: List[Int] = List(4)

scala> List(3) >> List()
val res2: List[Nothing] = List()

scala> List() >> List()
val res3: List[Nothing] = List()
```

MonadPlus and the guard Function

List comprehensions allow us to filter our output. For instance, we can filter a list of numbers to search only for numbers whose digit contains a 7:

```
ghci> [x | x <- [1..50], '7' `elem` show x]
[7,17,27,37,47]
```

We apply **show** to **x** to turn our number into a string, and then we check if the character '7' is part of that string. To see how filtering in list comprehensions translates to the list **monad**, we need to check out the **guard** function and the **MonadPlus** type class. The **MonadPlus** type class is for **monads** that can also act as **monoids**. Here is its definition:

```
class Monad m => MonadPlus m where
  mzero :: ma
  mplus :: ma -> ma -> ma
```

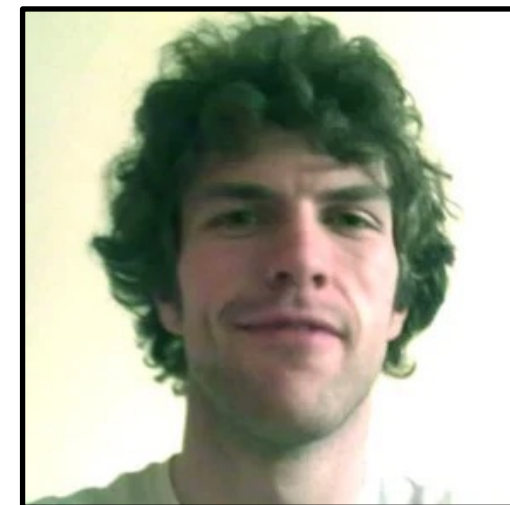
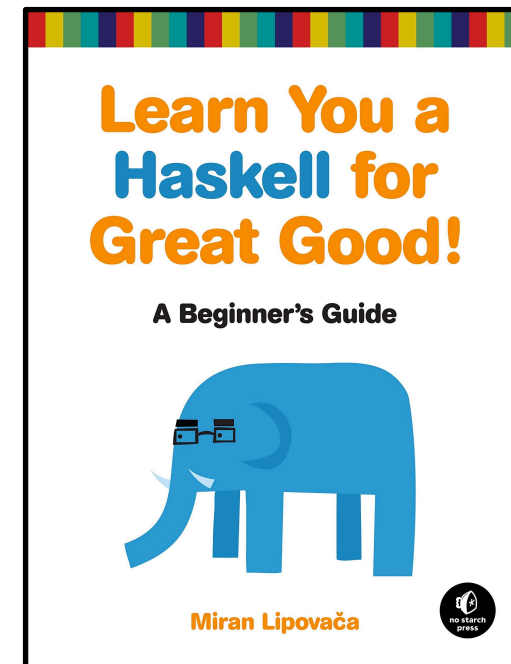
mzero is synonymous with **mempty** from the **Monoid** type class, and **mplus** corresponds to **mappend**. Because lists are **monoids** as well as **monads**, they can be made an instance of this type class:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

For lists, **mzero** represents a nondeterministic computation that has no results at all – a failed computation. **mplus** joins two nondeterministic values into one. The **guard** function is defined like this:

```
guard :: Bool -> m ()
guard True = return ()
guard False = mzero
```

guard takes a **Boolean** value. If that value is **True**, **guard** takes a **()** and puts it in a minimal default context that succeeds. If the **Boolean** value is **False**, **guard** makes a failed **monadic** value.



Miran Lipovača

Here it is in action:

```
ghci> guard (5 > 2) :: Maybe ()  
Just ()
```

```
ghci> guard (1 > 2) :: Maybe ()  
Nothing
```

```
ghci> guard (5 > 2) :: [()]  
[()]
```

```
ghci> guard (1 > 2) :: [()]  
[]
```

```
ghci> [x | x <- [1..50], '7' `elem` show x]  
[7,17,27,37,47]
```

This looks interesting, but how is it useful? **In the list monad, we use it to filter out nondeterministic computations:**

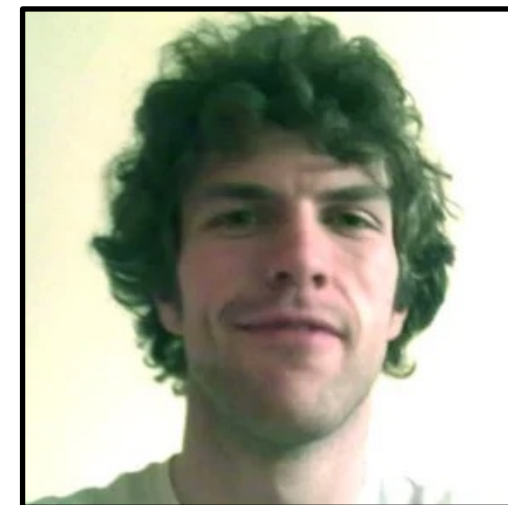
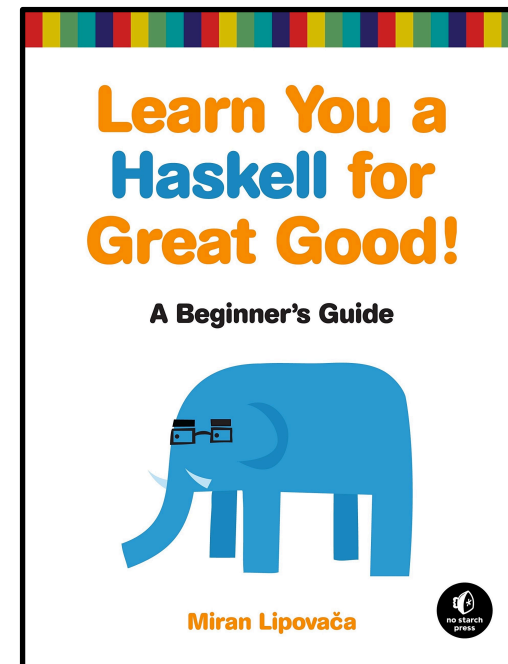
```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)  
[7,17,27,37,47]
```

same result

The result here is the same as the result of our previous list comprehension. How does guard achieve this? Let's first see how **guard** functions in conjunction with **>>**:

```
ghci> guard (5 > 2) >> return "cool" :: [String]  
["cool"]  
ghci> guard (1 > 2) >> return "cool" :: [String]  
[]
```

If **guard** succeeds, the result contained within it is the **empty tuple**. So then we use **>>** to ignore the **empty tuple** and present something else as the result. However, if **guard** fails, then so will the **return** later on, because feeding an empty list to a function with **>>=** always results in an empty list. **guard** basically says, "If this **Boolean** is **False**, then produce a failure right here. Otherwise, make a successful value that has a dummy result of **()** inside it." All this does is to allow the computation to continue.



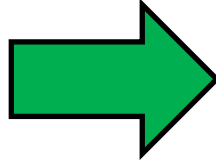
Miran Lipovača



Here we see the **desugaring** of the **Scala for comprehension** again, but we also **desugar** the equivalent **Haskell do expression**.



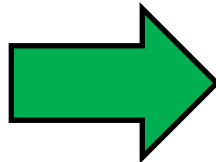
```
for
  queens <- placeQueens(k - 1)
  queen <- 1 to n
  if safe(queen, queens)
yield queen :: queens
```



```
placeQueens(k - 1) flatMap { queens =>
  (1 to n) withFilter { queen =>
    safe(queen, queens)
  } map { queen =>
    queen :: queens
  }
}
```



```
do
  queens <- placeQueens(k-1)
  queen <- [1..n]
  guard (safe queen queens)
return (queen:queens)
```



```
placeQueens(k-1) >>= \queens ->
  [1..n] >>= \queen ->
    guard (safe queen queens) >>
      return (queen:queens)
```



Note how, while the **desugared Scala** code introduces the use of **flatMap**, **withFilter** and **map**, the **desugared Haskell** code introduces the use of **>>=** (Haskell's equivalent of **flatMap**) and **>>**.

Note also how, in **Haskell**, the **guard** function is used in both the **do expression** and in its **desugared** equivalent.

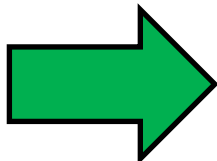


Same as on the previous slide, except that here the logic is simpler to understand and can be run at the **REPL**.

 @philip_schwarz



```
for
  nums <- List(List(1,2),List(3,4))
  num <- List(10,20)
  if (num > 15)
yield num :: nums
```

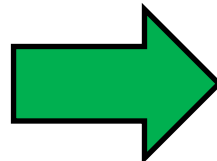


```
List(List(1,2),List(3,4)) flatMap { nums =>
  List(10,20) withFilter { num =>
    num > 15
  } map { num =>
    num :: nums
  }
}
```

List(List(20, 1, 2), List(20, 3, 4))



```
do
  nums <- [[1,2],[3,4]]
  num <- [10,20]
  guard (num > 15)
  return (num:nums)
```

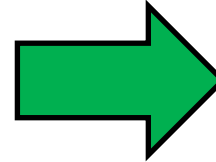


```
[[1,2],[3,4]] >>= \nums ->
  [10,20] >>= \num ->
  guard (num > 15) >>
    return (num:nums)
```

[[20,1,2],[20,3,4]]

guard lets
through **some**
elements

```
do
  nums <- [[1,2],[3,4]]
  num <- [10,20]
  guard (num > 15)
  return (num:nums)
```

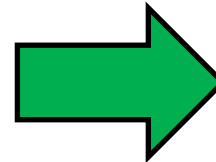


```
[[1,2],[3,4]] >>= \nums ->
[10,20] >>= \num ->
  guard (num > 15) >>
  return (num:nums)
```

List(List(20, 1, 2), List(20, 3, 4))

guard lets
through **all**
elements

```
do
  nums <- [[1,2],[3,4]]
  num <- [10,20]
  guard (True)
  return (num:nums)
```

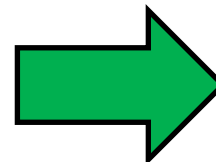


```
[[1,2],[3,4]] >>= \nums ->
[10,20] >>= \num ->
  guard (True) >>
  return (num:nums)
```

List(List(10, 1, 2), List(20, 1, 2), List(10, 3, 4), List(20, 3, 4))

guard lets
through **no**
elements

```
do
  nums <- [[1,2],[3,4]]
  num <- [10,20]
  guard (False)
  return (num:nums)
```



```
[[1,2],[3,4]] >>= \nums ->
[10,20] >>= \num ->
  guard (False) >>
  return (num:nums)
```

List()

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

...

...

Here is the previous expression rewritten in **do notation**:

```
listOfTuples :: [(Int,Char)]
listOfTuples do
  n <- [1,2]
  ch <- ['a','b']
  return (n,ch)
```

Do Notation and List Comprehensions

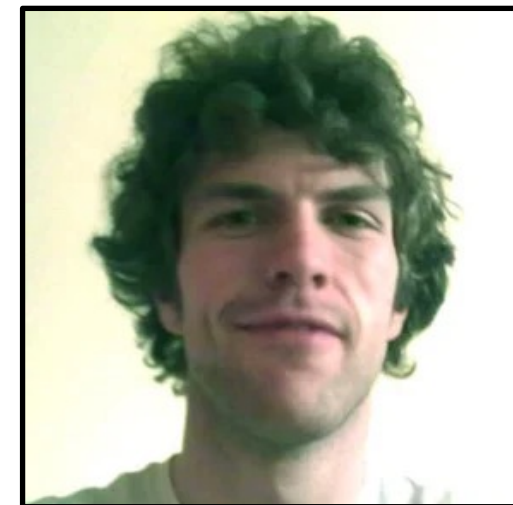
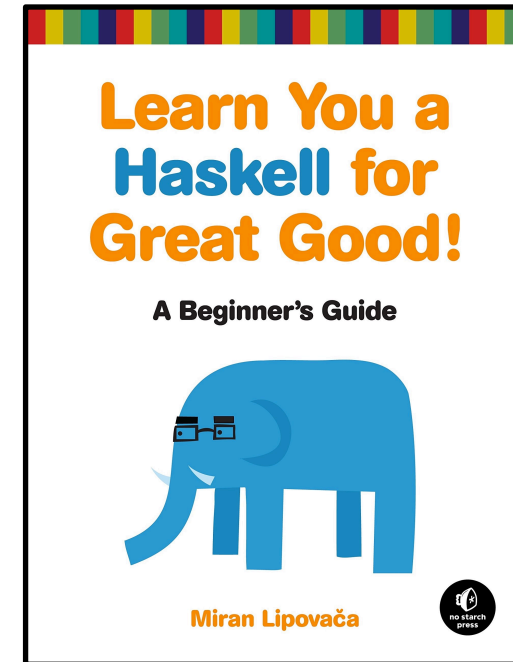
Using lists with **do notation** might remind you of something you've seen before. For instance, check out the following piece of code:

```
ghci> [(n, ch) | n <- [1,2], ch <- ['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Yes, **list comprehensions**! In our **do notation** example, **n** became every result from `[1,2]`. For every such result, **ch** was assigned a result from `['a','b']`, and then the final line put **(n,ch)** into a default context (a singleton list) to present it as the result without introducing any additional **nondeterminism**.

In this **list comprehension**, the same thing happened, but we didn't need to write **return** at the end to present **(n,ch)** as the result, because the output part of a **list comprehension** did that for us.

In fact, list comprehensions are just syntactic sugar for using lists as monads. In the end, **list comprehensions** and lists in **do notation** translate to using **>>=** to do computations that feature **nondeterminism**.



Miran Lipovača

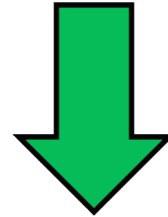
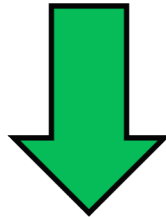


We saw earlier what the **Haskell do expression** looks like when it is **desugared**.

Desugaring the **list comprehension** produces the same result.

```
do queens <- placeQueens(k-1)
   queen <- [1..n]
   guard (safe queen queens)
   return (queen:queens)
```

```
[queen:queens |
 queens <- placeQueens(k-1),
 queen <- [1..n],
 safe queen queens]
```



```
placeQueens(k-1) >>= \queens ->
 [1..n] >>= \queen ->
   guard (safe queen queens) >>
   return (queen:queens)
```




As a recap, let's see again the translation of the whole program from **Scala** to **Haskell**.



```
def queens(n: Int): List[List[Int]] =
  def placeQueens(k: Int): List[List[Int]] =
    if k == 0
    then List(List())
    else
      for
        queens <- placeQueens(k - 1)
        queen <- 1 to n
        if safe(queen, queens)
      yield queen :: queens
  placeQueens(n)
```

```
def safe(queen: Int, queens: List[Int]): Boolean =
  val (row, column) = (queens.length, queen)
  val safe: ((Int,Int)) => Boolean = (nextRow, nextColumn) =>
    column != nextColumn && !onDiagonal(column, row, nextColumn, nextRow)
  zipWithRows(queens) forall safe
```

```
def onDiagonal(row: Int, column: Int, otherRow: Int, otherColumn: Int) =
  math.abs(row - otherRow) == math.abs(column - otherColumn)
```

```
def zipWithRows(queens: List[Int]): Iterable[(Int,Int)] =
  val rowCount = queens.length
  val rowNumbers = rowCount - 1 to 0 by -1
  rowNumbers zip queens
```

```
queens n = placeQueens n
  where
    placeQueens 0 = [[]]
    placeQueens k = [queen:queens |
      queens <- placeQueens(k-1),
      queen <- [1..n],
      safe queen queens]
```

```
safe queen queens = all safe (zipWithRows queens)
  where
    safe (r,c) = c /= col && not (onDiagonal col row c r)
    row = length queens
    col = queen
```

```
onDiagonal row column otherRow otherColumn =
  abs (row - otherRow) == abs (column - otherColumn)
```

```
zipWithRows queens = zip rowNumbers queens
  where
    rowCount = length queens
    rowNumbers = [rowCount-1,rowCount-2..0]
```



To conclude the first part of this slide deck, see below for a simple way of displaying the solutions to the **N-queens** problem.



There is lots to do in part two. We'll kick off by taking the **Scala** version of the program, and extending it so that it can display a single solution board as follows:

 @philip_schwarz



Functional Programming Principles in Scala

1.18K subscribers

6.3 Combinatorial Search Example

830 views • 11 Sept 2017

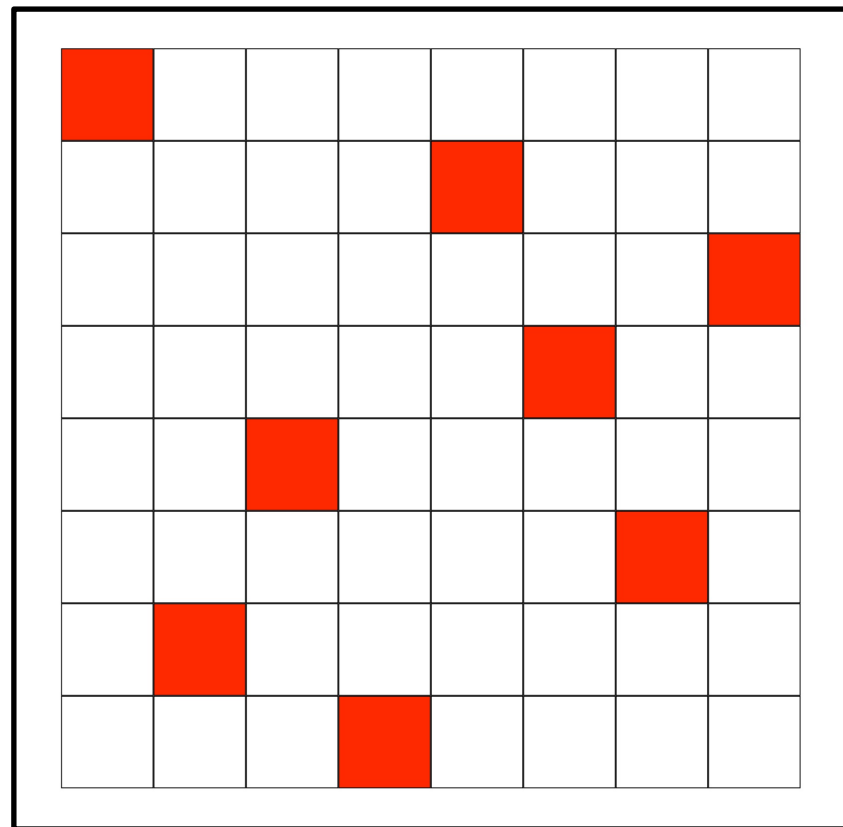
YouTube



```
def show(queens: List[Int]) = {
  val lines =
    for (col <- queens.reverse)
      yield Vector.fill(queens.length)("* ").updated(col, "X ").mkString
  "\n" + (lines mkString "\n")
}
```

queens(4) map show

```
> show: (queens: List[Int])java.lang.String
> res0: scala.collection.immutable.Set[java.lang.String]
| * * X *
| X * * *
| * * * X
| * X * * ", "
| * X * *
| * * * X
| X * * *
| * * X * ")
```



We'll then get the program to display, all together, the results of queens(N) for N = 4, 5, 6, 7, 8. Plus much more. See you then.