Sudoku Solver
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 10

Phil Wadler, Irene Vlassi-Pandi

Week 10 (18–22 Nov.)

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please send email to lambrose@ed.ac.uk if you cannot join your assigned tutorial.

Sudoku Solver

This tutorial will help you to write a solver for Sudoku puzzles. The exercise is adapted from Chapter 5 of ThinkingFunctionally with Haskell by Richard Bird (CUP, 2015).

Sudoku is played on a 9×9 grid, as shown in the diagram. The goal is to fill in each empty cell with a digit between 1 to 9, so that each row, column, and 3×3 box contains the numbers 1 to 9—hence no number should appear twice in any row, column, or box. Usually, Sudoku puzzles are designed to have a unique solution, but we will write a solver that returns a list of all possible solutions.

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

Figure 1: An easy Sudoku puzzle

1 Representing and printing Sudoku puzzles

We will represent a Sudoku puzzle as a matrix, where a matrix is a list of rows and each row is itself a list.

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

By convention, a row will always have nine elements and a matrix will always have nine rows, so a matrix will always represent a 9×9 grid. (We could use abstract data types to enforce this convention, see the optional problem at the end.)

We introduce a type to represent the contents of a cell.

```
type Digit     =  Char
```

By convention, a digit will be one of the characters from 1 to 9 or a blank. The following will be used later.

```
digits :: [Digit]
digits =  ['1'..'9']

blank :: Digit -> Bool
blank =  (== ' ')
```

An online Sudoku puzzle generator can be found at sudoku.com. It can generate puzzles at four levels of difficulty: easy, medium, hard, and evil. Here is an easy puzzle.

```
easy    :: Matrix Digit
easy    =  ["    345  ",
            "  89   3 ",
            "3    2789",
            "2 4  6815",
            "    4    ",
            "8765  4 2",
            "7523    6",
            " 1   79  ",
            "  942    "]
```

Exercise 1

Go to sudoku.com and generate three puzzles, one medium, one hard, and one evil, to add to the file.

We would like to print out puzzles in a more readable format.

```
*Main> put easy
-------------
|   | 34|5  |
|  8|9  | 3 |
|3  |  2|789|
-------------
|2 4|  6|815|
|   | 4 |   |
|876|5  |4 2|
-------------
|752|3  |  6|
| 1 |  7|9  |
|  9|42 |   |
-------------
```

We do this in a sequence of steps.

Exercise 2

Write a function that given a list of nine elements breaks it into three lists each containing three elements. It should work on elements of any type.

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

For example,

```
group "123456789"  ==  ["123", "456", "789"]
```

Exercise 3

Write a function which given a value and a list will intersperse the value before, between, and after each element of the list.

```
intersperse :: a -> [a] -> [a]
```

For example,

```
intersperse "|" ["123", "456", "789"]
  == ["|", "123", "|", "456", "|", "789", "|"]
```

Intersperse always takes a list of length n to a list of length $2 * n + 1$.

Exercise 4

Using `group` and `intersperse`, write a function

```
showRow :: String -> String
```

that converts one row of a Sudoku puzzle into a string for display. For example,

```
showRow "123456789" ==  "|123|456|789|"
```

Exercise 5

Again using `group` and `intersperse`, write a function

```
showGrid :: Matrix Digit -> [String]
```

that converts a list of rows of a Sudoku puzzle (each formatted for display) into a list of strings suitable for display. For example,

```
showGrid (replicate 9 "|123|456|789|") ==
  ["-------------",
   "|123|456|789|",
   "|123|456|789|",
   "|123|456|789|",
   "-------------",
   "|123|456|789|",
   "|123|456|789|",
   "|123|456|789|",
   "-------------",
   "|123|456|789|",
   "|123|456|789|",
   "|123|456|789|",
   "-------------"]
```

Exercise 6

Finally, using `unlines` and `putStrLn` from the standard library, write a function

```
put :: Matrix Digit -> IO ()
```

that will neatly display a Sudoku puzzle. For example,

```
*Main> put easy
-------------
|   | 34|5  |
|  8|9  | 3 |
|3  |  2|789|
-------------
|2 4|  6|815|
```

```
|   | 4 |   |
|876|5  |4 2|
-------------
|752|3  |  6|
| 1 |  7|9  |
|  9|42 |   |
-------------
```

2 Generating all possible solutions

We will represent a partially solved Sudoku puzzle by a matrix where each cell contains a list of the digits that might appear at that point.

Exercise 7

Write a function

```
choices :: Matrix Digit -> Matrix [Digit]
```

that converts a space to a list of all the digits, and otherwise converts a digit to a list containing just that digit. For example,

```
choices easy ==
 [["123456789","123456789","123456789","123456789","3",
   "4",        "5",        "123456789","123456789"],
  ["123456789","123456789","8",        "9",        "123456789",
   "123456789","123456789","3",        "123456789"],
  ["3",        "123456789","123456789","123456789","123456789",
   "2",        "7",        "8",        "9"],
  ["2",        "123456789","4",        "123456789","123456789",
   "6",        "8",        "1",        "5"],
  ["123456789","123456789","123456789","123456789","4",
   "123456789","123456789","123456789","123456789"],
  ["8",        "7",        "6",        "5",        "123456789",
   "123456789","4",        "123456789","2"],
  ["7",        "5",        "2",        "3",        "123456789",
   "123456789","123456789","123456789","6"],
  ["123456789","1",        "123456789","123456789","123456789",
   "7",        "9",        "123456789","123456789"],
  ["123456789","123456789","9",        "4",        "2",
   "123456789","123456789","123456789","123456789"]]
```

Exercise 8

Recall that the function `cp` (short for *cartesian product*) takes a list of length n of lists, and returns a list of lists of length n, where each element of the result contains one element from each list in the argument. The function is defined as follows.

```
cp :: [[a]] -> [[a]]
cp []       =  [ [] ]
cp (xs:xss)  =  [ y:ys | y <- xs, ys <- cp xss ]
```

For example,

```
cp ["ab","cd","efg"] ==
```

```
   ["ace","acf","acg","ade","adf","adg",
    "bce","bcf","bcg","bde","bdf","bdg"]
```

Observe that it satisfies the property:

```
length (cp xss) == product (map length xss)
```

By analogy with `cp`, write a function

```
expand :: Matrix [Digit] -> [Matrix Digit]
```

that produces a list of matrices of digits, with one matrix for each possible selection of digits from the lists in the original matrix. For example, working on smaller 2×2 matrices we have

```
expand [["12","34"],
        ["56","78"]] ==
  [["13",
    "57"],
   ["13",
    "58"],
   ["13",
    "67"],
   ["13",
    "68"],
   ["14",
    "57"],
   ["14",
    "58"],
   ["14",
    "67"],
   ["14",
    "68"],
   ["23",
    "57"],
   ["23",
    "58"],
   ["23",
    "67"],
   ["23",
    "68"],
   ["24",
    "57"],
   ["24",
    "58"],
   ["24",
    "67"],
   ["24",
    "68"]]
```

Exercise 9

Write a QuickCheck property that relates the length of the list returned by expand to the lengths of the lists of digits in its argument.

Exercise 10

Compute the length of the list of possible answers to the easy puzzle specified above. *Hint*: You will need to use the library function

```
fromIntegral :: Int -> Integer
```

to convert lengths to unbounded integers to avoid overflow in the answer. Assuming a computer can generate a trillion solutions in a second, how long would it take to consider all possible solutions to the easy puzzle? Compare this to the age of the universe, which is estimated at 13.7 billion years.

3 Rows, Columns, and Boxes

We will need to check that the numbers in each row, column, and box of a puzzle are distinct. For this reason, it is useful to write three functions from a matrix into a matrix:

```
rows, cols, boxs :: Matrix a -> Matrix a
```

Each of these maps the rows, columns, and boxes of a matrix into columns. Thus, if we consider matrices with distinct digits in each row, column, or box:

```
*Main> put byRow
-------------
|123|456|789|
|123|456|789|
|123|456|789|
-------------
|123|456|789|
|123|456|789|
|123|456|789|
-------------
|123|456|789|
|123|456|789|
|123|456|789|
-------------

*Main> put byCols
-------------
|111|111|111|
|222|222|222|
|333|333|333|
-------------
|444|444|444|
|555|555|555|
|666|666|666|
-------------
|777|777|777|
|888|888|888|
|999|999|999|
-------------

*Main> put byBox
-------------
```

```
|123|123|123|
|456|456|456|
|789|789|789|
-------------
|123|123|123|
|456|456|456|
|789|789|789|
-------------
|123|123|123|
|456|456|456|
|789|789|789|
-------------
```

Then we have:

```
rows byRow  ==  byRow
cols byCol  ==  byRow
boxs byBox  ==  byRow
```

Exercise 11

Define a function

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

that maps each row into a row. *Hint:* This is trivial.

Exercise 12

Define a function

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

that maps each column into a row. *Hint*: Look up the library function `transpose`, which we also used in Tutorial 4.

Exercise 13

Define a function

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

that maps each box into a row. *Hint*: using `group`, which we defined previously, and `cols`, which we just defined, and concat, which is in the standard library, we can make the transformations in Fig. 2, shown for a 4×4 rather than a 9×9 matrix.

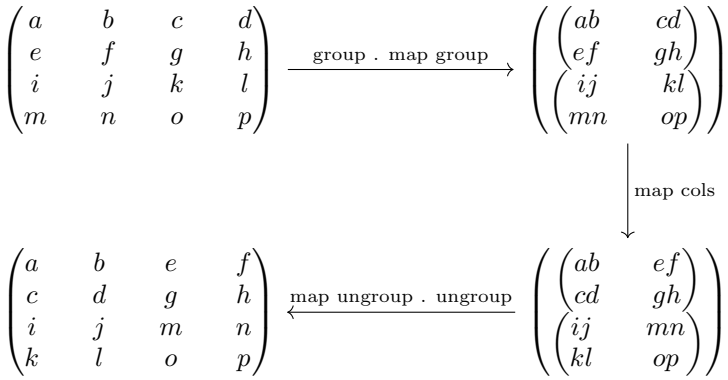

Figure 2: Boxs for 4×4 case

Exercise 14

Using the library function nub, write a predicate

```
distinct :: Eq a => Row a -> Bool
```

that holds if each item in a given row is distinct. *Hint*: this predicate was defined in Lecture 18.

Exercise 15

Write a function

```
valid :: Matrix Digit -> Bool
```

that holds if each row, column, and box in a Sudoku grid contains all the digits from 1 to 9.

Exercise 16

Observe that the function

```
simple :: Matrix Digit -> [Matrix Digit]
simple =  filter valid . expand . choices
```

will find all possible solutions to a given Sudoku puzzle. Given the answer to 10, is this a viable method?

Exercise 17

Observe that the following three remarkable properties hold:

```
rows . rows == id
cols . cols == id
boxs . boxs == id
```

We can describe this by saying the rows, cols, and boxs are *involutions*. Confirm these three properties using QuickCheck.

4 Pruning

Many Sudoku puzzles (such as the easy puzzle above) can be solved by the following technique. Recall that a partially solved puzzle is represented by the type

```
Matrix [Digit]
```

where each cell contains a list of digits that might appear in a solution. Some cells in the grid correspond to a definite value, and contain a singleton list. We can improve a solution by removing from the list of possible values in any cell any digit that appears in a singleton list in the same row, column, or box. We call such a step *pruning*. Sometimes, repeatedly applying pruning is enough to solve a puzzle.

Exercise 18

Write a function

```
pruneRow :: Row [Digit] -> Row [Digit]
```

that leaves any list consisting of a single digit unchanged, but removes from any other list any single digit that appears elsewhere in the row. For example,

```
pruneRow ["169","269","17","1678","3","4","5","26","1"]
  == ["69","269","7","678","3","4","5","26","1"]
```

Exercise 19

Inspired by the fact that rows, cols, and boxs are involutions, we can define a function

```
prune :: Matrix [Digit] -> Matrix [Digit]
pruneBy f  =  f . map pruneRow . f
```

Using `pruneBy`, define a function

```
prune :: Matrix [Digit] -> Matrix [Digit]
```

that performs pruning on each row, column, and box in the matrix.

Exercise 20

Write a function

```
many :: Eq a => (a -> a) -> a -> a
```

that takes a function `g` and value `x`, and repeatedly applies `g` to `x` until the value no longer changes. For example, say we define

```
close :: (Eq a, Ord a) => [(a,a)] -> [(a,a)]
close pairs  =  nub (sort (pairs ++
                    [ (x,z) | (x,y) <- pairs,
                      (y',z) <- pairs,
                      y == y' ]))
```

Then

```
close [(1,2),(2,3),(3,4)] ==
      [(1,2),(1,3),(2,3),(2,4),(3,4)]
close [(1,2),(1,3),(2,3),(2,4),(3,4)] ==
      [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
close [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)] ==
      [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

and so

```
many close [(1,2),(2,3),(3,4)] ==
      [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Exercise 21

Using the function

```
the :: [a] -> a
```

that takes a list of length one to its single element (and is undefined on other lists) write a function

```
extract :: Matrix [Digit] -> Matrix Digit
```

that takes a Sudoku configuration where every list contains a single digit into the puzzle consisting of those digits (and is undefined otherwise).

Exercise 22

Using `choices`, `prune`, `many`, and `extract`, write a function

```
solve :: Matrix Digit -> Matrix Digit
```

that solves any Sudoku puzzle that can be solved by repeated pruning. Which of the four given puzzles can be solved in this way?

5 Expanding a single cell

Given a partial solution of type `Matrix [Digit]`, there are three possibilities.

1. Every list of digits contains only one element. Then we can use extract to get the solution.
2. Some list of digits is empty. There there is no possible solution.
3. Some list of digits contains more than one element.

In the third case, we can consider the shortest such list, lets call it ds, and lets call its length n. For each d in ds, we can generate a new partial solution where ds is replaced by $[d]$, and then recursively try to solve each of the n resulting partial solutions.

For example, given the matrix of partial solutions:

```
[["9","1578","3","67","167","4","2","15678","5678"],
 ["4","178","6","5","12379","127","138","13789","789"],
 ["57","157","2","8","13679","167","135","1345679","5679"],
 ["238","238","9","267","2678","5","138","13678","4"],
 ["58","6","7","1","4","3","9","2","58"],
 ["1","23458","45","9","2678","267","358","35678","5678"],
 ["2356","123459","145","246","1256","8","7","59","259"],
 ["2567","1257","15","267","12567","9","4","58","3"],
 ["257","24579","8","3","257","27","6","59","1"]]
```

The earliest, shortest list containing more than one choice is "67" (the fourth element of the first row) and expanding it gives two simpler partial solutions.

```
[[["9","1578","3","6","167","4","2","15678","5678"],
  ["4","178","6","5","12379","127","138","13789","789"],
  ["57","157","2","8","13679","167","135","1345679","5679"],
  ["238","238","9","267","2678","5","138","13678","4"],
  ["58","6","7","1","4","3","9","2","58"],
  ["1","23458","45","9","2678","267","358","35678","5678"],
  ["2356","123459","145","246","1256","8","7","59","259"],
  ["2567","1257","15","267","12567","9","4","58","3"],
  ["257","24579","8","3","257","27","6","59","1"]],
 [["9","1578","3","7","167","4","2","15678","5678"],
  ["4","178","6","5","12379","127","138","13789","789"],
  ["57","157","2","8","13679","167","135","1345679","5679"],
  ["238","238","9","267","2678","5","138","13678","4"],
  ["58","6","7","1","4","3","9","2","58"],
  ["1","23458","45","9","2678","267","358","35678","5678"],
  ["2356","123459","145","246","1256","8","7","59","259"],
  ["2567","1257","15","267","12567","9","4","58","3"],
  ["257","24579","8","3","257","27","6","59","1"]]]
```

Exercise 23

Write a function

```
failed :: Matrix [Digit] -> Bool
```

that returns true if any of the lists of choices is empty.

Exercise 24

Write a function

```
solved :: Matrix [Digit] -> Bool
```

that returns true if all of the lists of choices contain exactly one digit.

Exercise 25

Write a function

```
shortest :: Matrix [Digit] -> Int
```

that returns the length of the shortest list of digits that has more than one choice.

Exercise 26

Write a function

```
break :: (a -> Bool) -> [a] -> ([a], [a])
```

that given a predicate and a list containing an element that satisfies the predicate, splits the list into two lists where the first is all the elements up to and not including the element which satisfies the predicate, and the second is the remainder, beginning with the element that satisfies the predicate. For example,

```
break (\ds -> length ds == 2)
["9","1578","3","67","167","4","2","15678","5678"]
== (["9","1578","3"],["67","167","4","2","15678","5678"])
```

Hint: this function is defined in the standard prelude.

Exercise 27

Using `shortest` and using `break` twice, write a function

```
expand1 :: Matrix [Digit] -> [Matrix [Digit]]
```

that performs the operation described at the beginning of this section: find the earliest, shortest list of choices containing more than one choice, and generate a new matrix for each choice in the list. *Hint:* We can break the matrix up into pieces using

```
(preMat, row:postMat) =  break (any p) mat
(preRow, ds:postRow)  =  break p row
```

where p is a predicate on lists of choices that returns true if its length is equal to a given length. We can then reassemble the array by computing

```
preMat ++ [preRow ++ [[d]] ++ postRow] ++ postMat
```

where d is a chosen digit from ds.

Exercise 28

Using `failed`, `solved`, `extract`, `many`, `prune`, and `expand1`, write a search program

```
search :: Matrix Digit -> [Matrix Digit]
```

that executes the `search` procedure described at the beginning of this section. Use it to solve all four sudoku puzzles.

6 Optional Material

Exercise 29

Use many to redefine reachable from Problem 7 in Tutorial 9.

Exercise 30

Using the declaration

```
data Matrix a = MkMat(Matrix a)
```

revise your Sudoku solver to be a module that defines an abstract data type. Guarantee that any client of the module can only define a matrix that has nine rows, each containing nine elements.