EDAF40: Lab 1
Building your own Sudoku puzzle, ver. 1.2

Adrian Roth and Jacek Malec

April 2, 2019

The goal of this lab is to get you to practise how to write Haskell code and use elementary Haskell functions that might prove useful for solving Assignment 1, Sudoku. In particular, we are going to work with a Sudoku board and look at how it can be represented as a Haskell data structure.

Part 1: How to program in Haskell

Step 1: Installing the Glasgow Haskell Compiler (GHC). If you are using the stationary computers in E house you can skip this step. Check out `https://www.haskell.org/downloads/` for installation instructions on various operating systems. The student system in E house runs GHC 7.10.3, but it might be wise to grab the most recent version.

Step 2: Using the interpreter (or repl) `ghci`.

- Open a terminal and write `ghci` (or open `WinGHCi` on Windows) where a command line is opened like in python or Matlab.
- Test to write Haskell code like `1 :  [1,2,3]`.
- or, `['a', 'b'] ++ ['c', 'd']`
- or, `map (2+) [1, 2, 3, 4]`
- Now try to look at the type of those functions by writing `:t map`, `:t (:)`, `:t (++)` and `:t map (2+)`. Discuss what does this mean?

Step 3: Loading files with code (modules) into the interpreter.

- Create a file `Sudoku.hs` with the following contents:

  ```
  module Sudoku where

  rows = "ABCD"
  cols = "1234"

  containsElem :: Eq a => a -> [a] -> Bool
  containsElem _ [] = False
  containsElem elem (x:xs)
    | elem == x = True
    | otherwise = containsElem elem xs
  ```

- In the `ghci` terminal write `:load Sudoku` (make sure that the file is in the same directory as the terminal running `ghci`).
- Try running `containsElem 1 [1,2,3]`, `containsElem 'a' "cde"` and other examples to see what happens.
- Discuss what the type specification means, `containsElem :: Eq a => a -> [a] -> Bool`. What is a and Eq?

Task: Write a function `cross :: [a] -> [a] -> [[a]]` which take two lists as input parameters and returns a list of lists of all combinations of the elements in the input lists.
Example `cross [1,2,3] [4,5] = [[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]`.
Hint: check out list comprehensions.

Part 2: Sudoku board

A contents of a Sudoku board will in this case be represented by a string of characters that are either a digit from `"0123456789"` or a '.'. An example Sudoku string is "0100200300040000" or ".1..2..3...4...." which are representations of the same 4x4 board shown below.

	1		
2			3
			4

Task 1: To have a nice representation of the input string we will first write a function to convert all '.' characters to '0' characters in the input string. Write a function `replacePointsWithZeros` which takes a string as input and returns a string with all '.' replaced with '0'. Try to also write the function type specification; look at the `containsElem` and `cross` functions for examples.

A Sudoku is built out of a number of squares where each square either has a value [1..9] or is empty. Each square can be denoted by a 2-character string, from now on referenced to as the square string, which is a letter for the row and a number for the column where the square is located, as in the following board.

A1	A2	A3	A4
B1	B2	B3	B4
C1	C2	C3	C4
D1	D2	D3	D4

A data structure to represent the Sudoku board in Haskell will be a list of tuples, `[(String, Int)]`, where the first element of a pair is a square string and the second is the value in this square (for empty square choose a suitable value).

Task 2: To create the data structure mentioned above a first step is to make a list of all the possible square Strings ["A1","A2",..,"D4"] for a 4x4 and 9x9 Sudoku boards (if you are creative you can also do 16x16).
Hint: use the cross function and the `rows` and `cols` variables (to be precise `rows` and `cols` are also functions which do not take any input parameters and return `String`s, though for clarity they are referred to as variables).

Task 3: Write a function `parseBoard` which takes a board String as input and returns a list of tuples representing the board as described above. The list of tuples will from here on be referenced to as the *grid*.
Hint: use the functions `replacePointsWithZeros`, `zip`, `map` and `digitToInt` together with the square string list. Remember to do `import Data.Char` to make `digitToInt` function to be in scope.

Part 3: Sudoku problem

The rest of the tasks are either made during the lab if there is time or afterwards before the next lab.

To make a smooth implementation of a Sudoku in Haskell one approach is to use the concepts *squares*, *units* and *peers*. *Squares* have already been introduced together with the square strings.
From the rules of Sudoku we know that each square has three *units*, one row unit, one column unit and one box unit. For example the square A1 has the row unit A, column unit 1 and box unit top left in the 4x4 board. Expressing it with the square strings, the units of A1 are [["A1","A2","A3","A4"],["A1","B1","C1","D1"], ["A1","A2","B1","B2"]].

Finally, the *peers* of square A1 are the squares in its units, without duplicates and without itself, i.e., in our case ["A2", "A3","A4","B1","C1","D1","B2"]. The peers of each square will be very useful in the implementation of a Sudoku validator and solver.

The next goal is to make a list of tuples where each tuple is a square string and a list of its peers, `peers :: [(String, [String])]`. In what follows we will create it for a 4x4 grid, by taking small steps.

Task 1: Calculate a variable `unitList :: [[String]]` of all the possible units (all rows, all cols and all boxes).
Hint: use the cross function and list comprehensions.

Task 2: Write a function `filterUnitList` which takes a square as input and use the `unitList` to return the three units which the square belongs to.
Hint: use the `containsElem` function.
Challenge: write this function in a point-free style.

Task 3: Calculate the variable `units` which is a list of tuples, where each tuple is a square string and its corresponding three units, `[(String, [[String]])]`.
Hint: use the `filterUnitList` function.

Task 4: Write function `foldList :: [[a]] -> [a]` which takes a list of lists and concatenates all sublists into a single list.
Challenge: write this function in a point-free style using higher order functions.

Task 5: Write function `removeDuplicates` which takes a list and removes all duplicates in that list, duh.

Task 6: Calculate the variable `peers` as presented above, remembering that the square string itself is not its own peer.
Hint: use the previous two implemented functions and the units variable.