EDAF95: Lab 1
Building your own Sudoku puzzle, ver. 2.02

Adrian Roth and Jacek Malec

March 23, 2023

The goal of this lab is to start practising HASKELL programming and write elementary functions that might prove useful for solving Assignment 1, Verify 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. Before the lab you should have attempted to do all preparation exercises and be able to explain your thoughts about a solution for each one.
All functions you are asked to write during the labs are meant to be used as inspiration for solving the assignments. If you find an approach that you think is more logical and easier to understand, we would be happy to hear about it!

To pass this lab the preparation tasks should be completed prior to the lab and then all tasks should be completed and presented to the teaching assistant.

Preparation exercises: Programming in HASKELL

Step 1: Installing the Glasgow HASKELL Compiler (GHC). If you are using the stationary computers in the E building you can skip this step. Check out `https://www.haskell.org/downloads/` for installation instructions on various operating systems. The student system in the E building runs GHC 7.10.3, but it might be wise to grab the most recent version.
Step 2: Using the interpreter repl[1] `ghci`. Open a terminal and write `ghci` (or open `WinGHCi` on Windows) where a command line is opened like in python or Matlab. Test by writing HASKELL code like

- `1 : [1,2,3]`.
- `(:) 1 [1,2,3]`.
- `['a', 'b'] ++ ['c', 'd']`
- `map (2+) [1, 2, 3, 4]`

[1] read-eval-print loop

- Now look at the type of some functions by writing `:t map`, `:t (:)`, `:t (++)` and `:t map (2+)`. Discuss what the type means?

Step 3: Loading files with code 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` repl 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]]` to use for the Sudoku 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 from lecture 2 or on the internet.

Part 1: Sudoku board

The input 4x4 Sudoku board will in this lab be represented by a string of characters that are either a digit [0..4] or a dot '.'. An example Sudoku string is "0100200300040000"or ".1..2..3...4...."which are both representations of the same 4x4 board shown below.

	1		
2			3
			4

Task 1: To have a uniform 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` that 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 9x9 Sudoku is represented by a number of squares where each square either has a value [1..9] or is empty. Each square can be represented with a 2 character string, from now on referenced to as the square string, which is a letter for each row and a number for the column as in the following board.
A data structure to represent the Sudoku board in HASKELL can be a list of pairs, `[(String, Int)]`, where the string stands for the square name and the integer is the value in this square (e.g., zero).

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

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"] in a 4x4 and then 9x9 Sudoku board (if you are creative you can also do a 16x16 board as well).
Hint: Use the `cross` function and the `rows` and `cols` variables.

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

Part 2: Sudoku problem

The rest of the tasks will also be preparation exercises for the next lab.

To make a smooth implementation of a Sudoku in HASKELL one approach is to use the concepts of *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 a 4x4 board. Or with the square strings the units of A1 are [["A1","A2","A3","A4"],["A1,"B1","C1","D1"], ["A1","A2","B1","B2"]]. At last the *peers* of square A1 are the units of A1 without duplicates and without itself, ["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 together with a list of its peers, `peers :: [(String, [String])]`, initially

for a 4x4 board, which we will attempt in smaller steps.

Task 1: Calculate a value `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 value `units` which is a list of tuples where each tuple is a square string together with 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, surprisingly, removes all duplicates in that list.

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