EDAF95: Lab 2
Building your own Sudoku puzzle, ver. 2.03

Adrian Roth and Jacek Malec

April 8, 2022

The goal of this lab is to continue with the Sudoku board and, in addition, work with `Hlint` and the algebraic data structure `Maybe`.

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 Part: Sudoku problem

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. Hint: Use the `containsElem` function.

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.

Part 1: *linting* your code

As written in the wikipedia article on LINT (`https://en.wikipedia.org/wiki/Lint_(software)`) a linter is a tool which can locate both syntax and stylistic errors in code. The `hlint` tool is adapted for HASKELL as you might have guessed and the usage is:
`hlint` *filename*,
(e.g., `hlint Sudoku.hs`). Even though your program is working, `hlint` will give you suggestions for how to properly express your code in HASKELL. As seen in the instructions for Assignment 1 it is assumed and strongly encouraged that you refine your code using this tool prior to submission.

After the linting you can probably feel your refactoring momentum and we should definitely not slow down. We continue by realising that there is a Prelude function called `elem` which does exactly the same thing as `containsElem`. Why not switch and remove `containsElem`. There is also a Prelude function that performs exactly what `foldList` does, find and replace.

The next thing we are concerned with is that the instructions for Assignment 1 say that we are only allowed to use prelude functions. Now most of you have used the `Data.Char` function `digitToInt` which means that you now should implement this function yourself. When you are done, do the same for all other non prelude functions you might have used in your code so far.
Hint: Remove all import statements and try to compile or load your code. The errors will show which functions you have used that are not included in Prelude.

Task 1: Lint your HASKELL code.

Part 2: The `Maybe` data type

The `Maybe` data type is commonly used in HASKELL when a function needs to handle possible error situations or exceptions.

Task 1: The `lookup` function. Look at its type and discuss what it does, and determine for which inputs it returns `Nothing`.

Task 2: Write a function
`fromMaybe :: a -> Maybe a -> a`
where the Maybe value is returned if it is `Just` and the first parameter otherwise.

Task 3: Write a function
`getPeers :: String -> [String]`
which returns the peers of the first parameter (the square string) using the `peers` value.
Hint: Use the functions `lookup` and `fromMaybe`.
Challenge: Write this function in a point-free style.

Task 4: Write a function
`justifyList :: [Maybe a] -> [a]`
which takes a list of `Maybe` objects and outputs a list of only the `Just` element values (without the constructor `Just`).

Task 5: Write a function
`lookups :: Eq a => [a] -> [(a, b)] -> [b]`
which is similar to the `lookup` function but takes a list of input values.
Hint: Use the functions `lookup` and `justifyList`.
Challenge: Write this function with only one point (one parameter).
Flipping Challenge: Write this function in point-free style. (Why is this called *flipping challenge*?)

Part 3: Sudoku Verifier, simple conflicts

In this part we will focus on the first problem of verifying a Sudoku, namely on considering the non empty squares. If a filled square has the same value as any of its peer squares the Sudoku is not consistent and the `verifySudoku` function should return `False`.

Remember that:

```
type Board = [(String, Int)]
type Sudoku = String
```

Task 1: Write a function
`validSquare :: (String, Int) -> [(String, Int)] -> Bool`
which checks if a single square tuple is valid in a Sudoku board.

Hint: If the square is empty we currently consider it consistent and otherwise use the functions `elem`, `lookups` and `getPeers` to see if the value is consistent.

Task 2: Write a function `validBoard` which checks whether all the squares in a board are valid.

Task 3: Write the function `verifySudoku` using previously implemented functions.

Task 4: Test your verifier with both consistent and inconsistent input data. Maybe prepare some test values inside your source code or create a test file. If you are adventurous, you may consider creating a unit test for your verifier.

Part 4: Sudoku verifier, blocking conflicts

At last we will consider the blocking conflicts introduced in Assignment 1, which might occur in a Sudoku. To find the blocking we need to now instead of the squares look at each unit and see if a unit is valid. As you remember a unit can either be a row, column or box where each square inside the unit is either empty or filled with a value. Now to check if there exists a blocking situation we can first for each empty square calculate all possible fill values without introducing simple conflicts to the square's peers. Secondly each unit will be checked for blocking in the following manner:

- For the squares with only one possible fill value in a unit, including the filled squares, no two squares have the same value (similar to the simple conflicts introduced above).
- For all squares in a unit verify that *every* possible square value `[1..4]` for 4x4 or `[1..9]` for 9x9 can be inserted into *at least one* of the squares.

Task 1: Write a function `reduceList` which from two input lists removes occurrences of elements in the second list from the first list.

Task 2: Revisit the function `validSquare` which is now rewritten to

```
validSquareNumbers ::
               (String, Int) -> [(String, Int)] -> (String, [Int])
```

This function returns a tuple where the second part of the tuple is a list of values which can be inserted in that square. For a filled square `(sq, v)` (`v` is value of non empty square) it is reasonable to return either a list with only that value (`(sq, [v])`), or an empty list (`(sq, [])`) if the filled square is invalid, as implemented in Part 3 of this lab.

Task 3: Change the function `validBoard` to `validBoardNumbers` which maps the `validSquareNumbers` function onto the full board.

Task 4: Write a function `validUnit :: [String] -> [(String, [Int])] -> Bool` which checks if a unit is valid. Here it is important to remember the second point in the list above (possibility to insert every value in a unit) so read it carefully and you will get it right.
Hint: The functions `and`, `elem`, `concat`, `lookups` can be used for this.

Task 5: Write a function `validUnits` which checks if all units in the variable `unitList` are valid for a Sudoku board.

Task 6: Update `verifySudoku`.

You are done. Now you should be able to finish Assignment 1.