EDAF40: Lab 2
Building your own Sudoku puzzle, ver. 1.2

Adrian Roth and Jacek Malec

April 12, 2019

The goal of this lab is to continue on the Sudoku board and, in addition, work with `Hlint` and the algebraic data structure `Maybe`. Remember that all functions written in this lab form one possible way of solving Assignment 1, but there are definitely many other approaches. If you find an approach which you think is more logical and easier to understand than our version, we would be happy to hear about it!

Preparation Part: Sudoku problem

To make a smooth implementation of a Sudoku in HASKELL one can use the concepts *squares*, *units* and *peers*. *Squares* have already been introduced together with the square strings in the previous lab. 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 is the units without duplicates and itself, ["A2", "A3","A4","B1","C1","D1","B2"]. The peers of each square will be very useful in the implementation of a Sudoku verifier and solver.

The next goal is to make a list of tuples where each tuple is a combination of a square string and a list of its peers, `peers :: [(String, [String])]`, which we will attempt in smaller steps.

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

Task 2: Write a function `filterUnitList` which takes a square String as input and use the `unitList` to return the three units which the square belongs to.
Hint: Use the `filter` and `containsElem` functions.

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 combination of a square string and its corresponding three units, `[(String, [[String]])]`.
Hint: Use the `filterUnitList` function together with either `map` and lambda function or list comprehension.

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

Task 5: Write function `removeDuplicates` which takes a list and removes all duplicates in that list, duh.
Hint: Use the `containsElem` function.
Challenge: Write this function in a point free style.

Task 6: Write function `delete :: Eq a => a -> [a] -> [a]` which removes the first occurrence of the first parameter in the second list parameter.

Task 7: Calculate the variable `peers` as presented above, remember that the square string itself is not its own peer.
Hint: Use the previous three implemented functions and the units variable; the implementation can look similar to the implementation of the units variable.

Part 1: *linting* your code (optional)

Note: Unfortunately, `hlint` is not (yet) available on LTH lab computers, meaning that this part would need to be done on your own machines, where you can install `hlint` without problems. Sorry for this miss. Jacek

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 Sudoku.hs`,
or whichever filename you use. Even though your program is working, `hlint` will give you tips and tricks for how to do "nice" programming in Haskell. As seen in the instructions for Assignment 1 is it 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 definitly 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`. Wait a minute, there is a Prelude function that performs exactly what `foldList` does as well, find and replace.

The next thing we are concerned about is that the instructions for Assignment 1 says 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 with that 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.

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. It can either have the value `Just a` where `a` is any type or `Nothing` (typically something has gone wrong if a `Maybe` type has the value `Nothing`).

Task 1: Check out the `lookup` function. Look at its type and discuss what it does and 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 square string using the peers variable.
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` data type objects and outputs a list of 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.

Part 3: Sudoku Verifier, simple conflicts

In this part we will focus on the first problem of verifying a Sudoku by only 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`.

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 value of a square is zero 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 if all the squares in a board are valid.

Task 3: Write the greatest, and possibly shortest, function `verifySudoku` using appropriate functions previously implemented.

Task 4: Test your verifier with both consistent and inconsistent input data. Maybe write some test variables inside your source code.

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 will first calculate all the values which can be filled into an empty square without any conflict to the squares peers. We do this for all empty squares. Secondly, each unit will be checked for blocking in the following manner, where the squares of the unit are considered:

- For the filled squares there is a maximum of one square with each value in a unit.
- For all squares in a unit it is checked that *every* possible square value `[1..4]` for 4x4 or `[1..9]` for 9x9 can either be inserted into *at least one* of the empty squares or is already existing in the filled 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 will now be 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 not equal to zero) either return a list with only that value (`((sq, [v]))`) or an empty list (`((sq, []))`) if the filled square is invalid as implemented in Part 3 or 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 so read it until you find a reasonable explanation of how to proceed and try that. If it does not work try a different interpretation of it. The information is there even though it is not seen at first glance.
Hint: The functions `and`, `elem`, `concat`, `lookups` can be used for this.

Task 5: Write a function `validUnits` which checks if all units in your variable `unitList` are valid.

Task 6: Change in verifySudoku to use `validUnits` instead of `validBoard`.