COMP1100 Assignment 3

Sudoku

Report

ANU CECS

Name:	Aditya Sharma
Student Number:	u6051965
Course:	COMP1100
Date:	05/05/17
Collaborators:	Arham Qureshi - u6378881 Nathaniel McGrath - u5643010

A mathematically driven assessment task, Sudoku proved to be a definite difficulty in comparison to past COMP1100 assessments. The functions given required modest capability in most basic aspects of Haskell and therefore were a good test for all concepts taught so far.
The given hints were used in every function where they were applicable, and any helper functions created were for purposes of either cleaner code in the main function they were to be used, or to do a task for which there were no prelude functions assigned.

All imports besides the ones given at the start have been used to cut down the amount of code that would otherwise be required or to avoid cluttering the program with helper functions.

_____

```
allBlanks :: Sudoku
```

allBlanks required 9 lists of 9 nothing in each. Replicating nothing 9 times and replicating this result 9 times then converting the overall result to a Sudoku type gave us what we needed. The hinted replicate function was useful.

```
isSudoku :: Sudoku -> Bool
```

Simply check if the list of mapped length values for each list is equal to 9 and check if this given list of length values has a length of 9. If this returns true, we have a 9x9 sudoku. The hinted all function was utilized.

```
noBlanks :: Sudoku -> Bool
```

Requiring a little bit more thinking compared to the past two, this one required a way to check if all values were nothing or not. Filtering out the nothing values and then checking the length of the resulting list for every list in the Sudoku proves to be a valid method. If there are truly no nothing values, then each list will be of length 0.

```
printSudoku :: Sudoku -> IO ()
```

See `converter` below. putStrLn is used to print out the resulting values to console, however the map function does not work on putStrLn due to conflicting monad types, therefore a different function, mapM_ is used. This function prints user defined types to IO.

```
converter :: Sudoku -> [[Char]]
```

A converter function written for printSudoku, this function is used to create a readable representation of a given Sudoku. There is no function which gets the Integer types for maybe values from a list then converts them all to a string. A possible implementation was to convert all the values to a char type using intToDigit, however we needed the 0s to be '.', therefore Data.String.Utils was imported and the replace function was used to easily accomplish this task. This converter function is then fed into printSudoku, resulting in shorter and cleaner code.

```
fromString :: String -> Sudoku
```

See `maybeMapper` below. In order to complete this function, an effective opposite to the converter function needed to be created. Using this helper makes our lives a lot easier as the resulting row cells are effectively sudokus.

```
maybeMapper :: String -> Row Cell
```

maybeMapper is effectively a reverse of the converter function, taking in the list representation and converting each given char type into its respective digit representation. The resulting Row Cell type is fed into fromString. There was no function in the prelude which did what fromString needed, therefore this function was user-defined.

```
toString :: Sudoku -> String
```

toString and printSudoku have a lot in common – they both require canonical string representations of the given Sudoku to complete the tasks they are needed for. The converter function defined prior came in use here also, as we could use the resulting char types as a string value after concatenation.

```
rows :: Matrix a -> [Block a]
```

This function (and the cols and boxes functions following it) seemed daunting at first but is simply used to change the type of a Sudoku, as the given definition for Sudoku is already is rows. Therefore we return the same result which has its type cast.

```
cols :: Matrix a -> [Block a]
```

Also seemingly difficult, this function used one of the hinted functions given to us – transpose.

```
boxs :: Matrix a -> [Block a]
```

The first major road-bump in this assessment, the logic required to generate boxes was somewhat difficult to wrap one's head around. Thanks to peer review and explanation, the best method to do this proved to be to represent the given line splits in a Sudoku, but in code. First we split into 3 to achieve the groups between horizontal lines, then we transpose these into columns and split the result into chunks of 3 to give our vertical splits and representations for boxes. Easy in theory, but difficult to implement as it was the first function of its nature.

```
okBlock :: Block Cell -> Bool
```

An easy implementation which required checking if each element of a given block list was repeated. notElem proved to be a useful import function.

```
prop_Sudoku :: Sudoku -> Bool
```

A test function, the most straightforward implementation of this was to check if all boxes, rows, colums and cells were of length 9. IsSudoku already checked off a few of these cases for uses and was therefore used. The rest was to simply check the length of each respective list.

```
okSudoku :: Sudoku -> Bool
```

The easiest implementation for this was to map okBlock to rows, cols, and boxes, however blockCheck was defined for this purpose and used for cleaner code.

```
blockCheck :: [Block Cell] -> Bool
```

see okSudoku.

```
blank :: Sudoku -> Pos
```

The second difficult implementation, the easiest way to do this effectively was to first get the indexes of the first nothing in every row (see xChecker), map this to each row, meaning that the indexes of the resulting rows will be y positions (see yPos), and if invalid, jump to the next valid Nothing value with (xIterator). 11 is used universally throughout these functions as somewhat of an error number. In yPos, if the index was not 0, the function would add 1 to it to achieve the actual y position, however if the index was 8, the position would be 9 and this would be wrong. The resulting function to check this is (yCheck).

```
(!!=) :: [a] -> (Int, a) -> [a]
```

Simple take and drop implementation in respect to index.

```
update :: Sudoku -> Pos -> Int -> Sudoku
```

Use (!!=) on the x position list, then on the overall Sudoku.

```
solve :: String -> [String]
```

Simple backtracking used with lambda for simplicity. The function goes over each value and brute forces this for each cell. Base implementation. This is defined in solver.

doctest results:

```
Adis-MacBook-Air:assignment3 Adi$ doctest Sudoku.hs
Examples: 16  Tried: 16  Errors: 0  Failures: 0
```

easy.txt

```
real    2m8.448s
user    2m4.399s
sys     0m2.612s
```

The program unfortunately failed to complete hard.txt effectively as the backtrack algorithm took too long – same with Sudoku17.txt.

An interesting assessment, this really tested each individuals skills with Haskell and its notation, and resulted in an improved ability to program in this language and to thing like a programmer – recursion, lazy programming, evaluation, and everything else under the moon which would improve one's logical competency – a rewarding task.