Assignment 3: Sudoku

Name:	Ernest Kun Chi Kwan
Student Number:	u6381103
Course:	COMP1130
Date:	26/05/2017

Introduction

In this assignment, we're asked to write a program to solve the puzzle Sudoku (数独). Our program uses backtracking, a brute-force search, with heuristics to guide the search to the solution more quickly. Propagation is used when solving the Sudoku to reduce the number of brute-force searches.

In the first part of the assignment, the program picks the first blank spot and tries to recursively solve the Sudoku 9 times, trying each number from 1 to 9 on the cell, until it finds a solution. A solution is found when there are no blanks left and every block (row, column and 3x3 box) contains no repeated digit.

In extensions, the program would pick a blank that has the least candidate values in the row, column and box. Propagation is applied to produce a simpler Sudoku which is closer to the final solution. In addition, the program only uses values missing from the row, column and box to do a brute-force search instead of all integers from 1 to 9.

Implementation

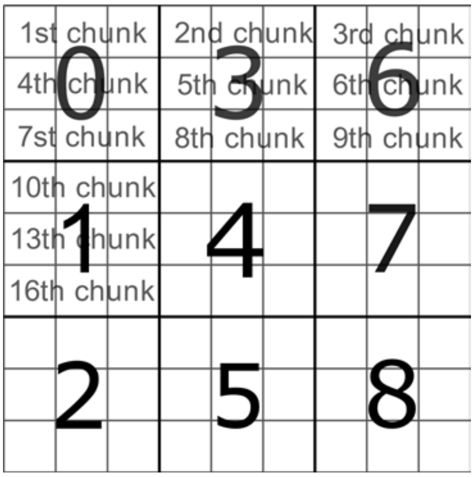
Imports

Data.List is imported as a Sudoku is modelled as a Matrix of Cells in our program, which is a list of lists of `Maybe Int`. For example, I used replicate from Data.List to create the allBlanks Sudoku which is 9 lists of 9 `Nothing`. Data.Char is imported for conversion between Sudoku and String, which is needed for input/output. Data.List.Split is imported for the `chunksOf` function which splits the list into pieces with length n. Data.Maybe is imported which makes working with `Maybe` types easier. For example, fromJust is used to converts a `Maybe Int` into `Int`.

Doctest and quickCheck

An instance for generating Arbitrary Sudokus is provided, which allows us to check other functions using `doctest`. For example, functions `fromString` and `toString` can be checked with the property `fromString (toString s) == s`. The function `prop_Sudoku` checks if the dimensions of the rows, columns and boxes are correct, and can be done by `quickCheck prop_Sudoku.`

Extracting the blocks of a matrix



Extracting the rows is trivial, as it is simply given by the original matrix. The columns are extracted by taking the transpose of the matrix, with a built-in function. The 3x3 boxes are extracted in the following way.

Firstly, the matrix is split into chunks of 3. The helper function takes the chunks in the required order, i.e. the 1st, 4th and 7th chunk which would make up the first box, the 10th, 13th and 16th chunk which would make up the second box, and so on. Then we build the list of blocks by concatenating every 3 chunks.

Finding blanks

I originally wrote a naïve `blank` function that simply goes through the Sudoku from the top and finds the first blank cell.

I wrote the function `prop_Blank` so I can use `quickCheck prop_Blank` to make sure that `blank` returns the position of a `Nothing` cell.

For extension 1, I first wrote a `blank` function that picks the block with the least number of blanks and then find the first blank in that block. The idea is to finish the supposedly easiest block first, then move on to other blocks with the least number of blanks.

```
blank' :: Sudoku -> Pos
blank' (Sudoku s)
    | minCols <= minRows && minCols <= minBoxs
    = case elemIndex minCols (countBlanks (cols s)) of
        Nothing -> error "cant find minCols in blank'"
        Just j  -> colhelper (cols s !! j) 0 j
    | minRows <= minCols && minRows <= minBoxs
    = case elemIndex minRows (countBlanks (rows s)) of
        Nothing -> error "cant find minRows in blank'"
        Just i  -> rowhelper (rows s !! i) i 0
    | otherwise = case elemIndex minBoxs (countBlanks (boxs s)) of
        Nothing -> error "cant find minBoxs in blank'"
        Just i  -> boxhelper (boxs s !! i) ((i `mod` 3) * 3) ((i `div` 3) * 3) 0
    where
        colhelper :: Block Cell -> Int -> Int -> Pos
        colhelper [] _ _ = error "empty col in colhelper"
        colhelper (x:xs) i j = case x of
            Nothing -> (i, j)
            _       -> colhelper xs (i+1) j

        rowhelper :: Block Cell -> Int -> Int -> Pos
        rowhelper [] _ _ = error "empty row in rowhelper"
        rowhelper (x:xs) i j = case x of
            Nothing -> (i, j)
            _       -> rowhelper xs i (j+1)

        boxhelper :: Block Cell -> Int -> Int -> Int -> Pos
        boxhelper [] _ _ _ = error "empty box in boxhelper"
        boxhelper (x:xs) i j k = case x of
            Nothing -> (i + (k `div` 3), j + (k `mod` 3))
            _       -> boxhelper xs i j (k+1)
         countBlanks = map (length . filter isNothing)
         minElem :: [Int] -> Int
         minElem a = case a of
             []  -> 10
             x:xs
                 | x > 0 -> min x (minElem xs)
                 | otherwise -> minElem xs
         minCols = minElem (countBlanks (cols s))
         minRows = minElem (countBlanks (rows s))
         minBoxs = minElem (countBlanks (boxs s))
```

Then I decided to experiment with only considering rows and columns. I deleted `minBoxs` and `boxhelper` so the only comparison left is between `minCols` and `minRows`. That gave me a faster result.

After that, I decided to write another `blank` function which first calculates the number of blanks in every row and column, and then pick a blank spot which has the lowest total of candidate values. This gives a similar runtime. So I also tried experimenting with calculating the number of blanks of every row, column and box. It gives slightly lower results on easy Sudokus, but much faster on hard Sudokus. Hence, I end up using this version as the final version.

Solving Sudokus

I wrote the first version of `solve` function as follows.

```
solve :: String -> [String]
solve [] = []
solve str = case fromString str of
    s -> map toString (solve' s)
    where
        solve' :: Sudoku -> [Sudoku]
        solve' s
            | not (okSudoku s) = []
            | noBlanks s = [s]
            | otherwise        = do
                i <- [1..9]
                let s' = update s (blank s) i
                solve' s'
```

When a valid Sudoku is taken which contains blanks, it tries every integer from 1 to 9 on the position which is given by the `blank` function. I then applied constraints to the values for the function to try with the function `missingValues`, to reduce the number of blind guesses. It will only brute-force searches on the values that are not on the same row, column and box.

The function `propagate` finds blocks with only 1 or 2 blanks and apply rules that humans often use to solve Sudokus. For example, if there are 2 blanks on a row, and a value is on one of the column of the blanks, then the value must be on the other blank, since there can’t be two of the same values on the same column. Hence, this will produce a Sudoku closer to the solution.

References

Cs.anu.edu.au. (2017). *Assignment 3: 数独 / Sudoku / 'numbers alone' / 'number place' - Programming as Problem Solving (including Advanced)*. [online] Available at: https://cs.anu.edu.au/courses/comp1100/assignments/03/ [Accessed 26 May 2017].

En.wikipedia.org. (2017). *Sudoku*. [online] Available at: https://en.wikipedia.org/wiki/Sudoku [Accessed 26 May 2017].

Hackage.haskell.org. (2017). *Data.List*. [online] Available at: https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-List.html [Accessed 26 May 2017].

Hackage.haskell.org. (2017). *Data.List.Split*. [online] Available at: https://hackage.haskell.org/package/split-0.2.3.2/docs/Data-List-Split.html [Accessed 26 May 2017].

Hackage.haskell.org. (2017). *Data.Maybe*. [online] Available at: http://hackage.haskell.org/package/base-4.9.1.0/docs/Data-Maybe.html [Accessed 26 May 2017].

Sudokudragon.com. (2017). *Sudoku Puzzle solving strategies*. [online] Available at: http://www.sudokudragon.com/sudokustrategy.htm [Accessed 26 May 2017].