Lab Assignment 3 -- Sudoku

Some notes:

- Remember that you have to work in pairs. **Only groups of size 2 are allowed to submit!** Submissions by only 1 person or 3 or more persons will not be accepted by the Fire system.
- **When you are done, please submit your solution using the Fire system.**
- If you are stuck on this assignment, please read the page on **Getting Help** carefully!

Good luck!

Lab Assignment 3 -- Sudoku

In this Lab Assignment, you will design a Haskell program that will be able to solve Sudokus, a popular logical puzzle originating from Japan.

Assignments and Deadlines

There are 6 regular assignments as part of this Lab: A, B, C, D, E, and F. The lab consists (again) of two parts.

For submission, assignments A, B, C and D are called **Lab 3A**.

Assignments E and F are called **Lab 3B**.

Deadlines for each of these parts are given on the **home page**. There are also **extra assignments**. You can choose freely whether to do one of these. Those are just for fun.

Hints

Some assignments have hints. Often, these involve particular standard Haskell functions that you could use. Some of these functions are defined in modules that you have to import yourself explicitly. You can use the following resources to find more information about those functions:

- **Hoogle**, the library function search engine
- **Haskell Library Structure**, all standard libraries, for you to browse (scroll down to heading "Modules")
- **A Tour of the Haskell Prelude** shows all standard Haskell functions that you get without importing any module

We encourage you to actually go and find information about the functions that are mentioned in the hints!

Sudokus

Sudoku is a logic puzzle originating in Japan. In the West it has caught on in popularity enormously over the last five years or so. Most newspapers now publish a daily Sudoku puzzle for the readers to solve.

A Sudoku puzzle consists of a 9x9 grid. Some of the cells in the grid have digits (from 1 to 9), others are blank. The objective of the puzzle is to fill in the blank cells with digits from 1 to 9, in such a way that every row, every column and every 3x3 block has exactly one occurrence of each digit 1 to 9.

Here is an example of a Sudoku puzzle:

3	6			7	1	2		
	5					1	8	
		9	2		4	7		
				1	3		2	8
4			5		2			9
2	7		4	6				
		5	3		8	9		
	8	3					6	
		7	6	9			4	3

And here is the solution:

3	6	4	8	7	1	2	9	5
7	5	2	9	3	6	1	8	4
8	1	9	2	5	4	7	3	6
5	9	6	7	1	3	4	2	8
4	3	1	5	8	2	6	7	9
2	7	8	4	6	9	3	5	1
6	4	5	3	2	8	9	1	7
9	8	3	1	4	7	5	6	2
1	2	7	6	9	5	8	4	3

In this lab assignment, you will write a Haskell program that can read in a Sudoku puzzle and solve it.

More Information

If you want to read more about Sudokus, here are a few links:

- The **Daily Sudoku** has examples and explanations
- **Wikipedia on Sudoku**
- **sudoku.com** has examples and explanations
- **sudoku.com.au** has sudoku puzzles that you can solve online

---

Modelling Sudokus

To implement a Sudoku-solving program, we need to come up with a way of modelling Sudokus. A Sudoku is a matrix of digits or blanks. The natural way of modelling a matrix is as a list of lists. The outer list represents all the rows, and the elements of the list are the elements of each row. Digits or blanks can be represented by using the Haskell **Maybe** type. Digits are simply represented by **Int**.

Summing up, a natural way to represent Sudokus is using the following Haskell datatype:

```
data Sudoku = Sudoku [[Maybe Int]]
```

Since it is convenient to have a function that extracts the actual rows from the Sudoku, we actually use the following equivalent datatype definition:

```
data Sudoku = Sudoku { rows :: [[Maybe Int]] }
```

For example, the above Sudoku puzzle has the following representation in Haskell:

```
example :: Sudoku
example =
  Sudoku
    [ [Just 3, Just 6, Nothing,Nothing,Just 7, Just 1, Just 2, Nothing,Nothing]
    , [Nothing,Just 5, Nothing,Nothing,Nothing,Nothing,Just 1, Just 8, Nothing]
    , [Nothing,Nothing,Just 9, Just 2, Nothing,Just 4, Just 7, Nothing,Nothing]
    , [Nothing,Nothing,Nothing,Nothing,Just 1, Just 3, Nothing,Just 2, Just 8]
    , [Just 4, Nothing,Nothing,Just 5, Nothing,Just 2, Nothing,Nothing,Just 9]
    , [Just 2, Just 7, Nothing,Just 4, Just 6, Nothing,Nothing,Nothing,Nothing]
    , [Nothing,Nothing,Just 5, Just 3, Nothing,Just 8, Just 9, Nothing,Nothing]
    , [Nothing,Just 8, Just 3, Nothing,Nothing,Nothing,Nothing,Just 6, Nothing]
    , [Nothing,Nothing,Just 7, Just 6, Just 9, Nothing,Nothing,Just 4, Just 3]
    ]
```

Now, a number of assignments follows, which will lead you step-by-step towards an implementation of a Sudoku-solver.

Some Basic Sudoku Functions

To warm up, we start with a number of basic functions on Sudukos.

Assignment A

A1. Implement a function

```
allBlankSudoku :: Sudoku
```

that represents a Sudoku that only contains blank cells (this means that no digits are present).

Do not use copy-and-paste programming here! Your definition does not need to be longer than a few short lines.

A2. The Sudoku type we have defined allows for more things than Sudokus. For example, there is nothing in the type definition that says that a Sudoku has 9 rows and 9 columns, or that digits need to lie between 1 and 9. Implement a function

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

that checks if all such extra conditions are met by the given Sudoku.

Examples:

```
Sudoku> isSudoku (Sudoku [])
False
Sudoku> isSudoku allBlankSudoku
True
Sudoku> isSudoku example
True
Sudoku> isSudoku (Sudoku (tail (rows example)))
False
```

A3. Our job is to solve Sudokus. So, it would be handy to know when a Sudoku is solved or not. We say that a Sudoku is solved if there are no blank cells left to be filled in anymore. Implement the following function:

```
isSolved :: Sudoku -> Bool
```

Note that we do not check here if the Sudoku is a **valid** solution; we will do this later. This means that any Sudoku without blanks (even Sudokus with the same digit appearing twice in a row) is considered solved by this function!

Hints

To implement the above, use list comprehensions! Also, the following standard Haskell functions might come in handy:

```
replicate :: Int -> a -> [a]
length    :: [a] -> Int
and       :: [Bool] -> Bool
```

To help you get started, here is a file that you can use:

- **Sudoku.hs**, with some definitions that help you get going with Assignments A, B and C.

Reading and Printing Sudokus

Next, we need to have a way of representing Sudokus in a file. In that way, our program can read Sudokus from a file, and it is easy for us to create and store several Sudoku puzzles.

The following is an example text-representation that we will use in this assignment. It actually represents the example above.

```
36..712..
.5....18.
..92.47..
....13.28
4..5.2..9
27.46....
..53.89..
.83....6.
..769..43
```

There are 9 lines of text in this representation, each corresponding to a row. Each line contains 9 characters. A digit 1 -- 9 represents a filled cell, and a period (.) represents a blank cell.

Assignment B

B1. Implement a function:

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

that, given a Sudoku, creates instructions to print the Sudoku on the screen, using the format shown above.

Example:

```
Sudoku> printSudoku allBlankSudoku
.........
.........
.........
.........
.........
.........
.........
.........
.........
Sudoku> printSudoku example
36..712..
.5....18.
..92.47..
....13.28
4..5.2..9
27.46....
..53.89..
.83....6.
..769..43
```

B2. Implement a function:

```
readSudoku :: FilePath -> IO Sudoku
```

that, given a filename, creates instructions that read the Sudoku from the file, and deliver it as the result of the instructions. You may decide yourself what to do when the file does not contain a representation of a Sudoku.

Examples:

```
Sudoku> do sud <- readSudoku "example.sud"; printSudoku sud
36..712..
.5....18.
..92.47..
....13.28
4..5.2..9
27.46....
..53.89..
.83....6.
..769..43
Sudoku> readSudoku "Sudoku.hs"
Program error: Not a Sudoku!
```

Hints

To implement the above, you will need to be able to convert between characters (type **Char**) and digits/integers (type **Int**). The standard functions **chr** and **ord** (import the module **Data.Char**) will come in handy here. Think about the following problems:

- Given a character representing a digit, for example '3' :: Char. How do you compute the integer value 3?
- Given a digit represented as an integer, for example 3 :: Int. How do you compute the character '3'?

The constant value **ord '0'** will play a central role in all this.

Here are some functions that might come in handy:

```
chr        :: Int -> Char
ord        :: Char -> Int
putStr     :: String -> IO ()
putStrLn   :: String -> IO ()
sequence_  :: [IO a] -> IO ()
readFile   :: FilePath -> IO String
lines      :: String -> [String]
```

Here are some example Sudoku-files that you can download and use:

- **example.sud**, containing the above example.
- **sudokus.zip**, a ZIPped collection of sudokus, both easy and hard ones. The easy ones should all be solvable by your final program within minutes; the hard ones will probably take a very long time (unless you do extra Assignment X and/or Y)!.

Generating Sudokus as Test Data

Finally, we need to be able to test properties about the functions related to our Sudokus. In order to do so, QuickCheck needs to be able to generate arbitrary Sudokus.

Let us split this problem into a number of smaller problems. First, we need to know how to generate arbitrary cell values (of type **Maybe Int**). Then, we need to know how to generate 81 such cells, and compose them all into a Sudoku.

Assignment C

C1. Implement a function:

```
cell :: Gen (Maybe Int)
```

that, contains instructions for generating a Sudoku cell. You have to think about the following:

- Cells either contain a digit between 1 and 9 (for example **Just 3**) or are empty (**Nothing**),
- We would like our generated Sudokus to resemble realistic Sudoku puzzles. Therefore, the distribution should be around 10% probability non-empty cells vs. 90% probability for empty cells. (This is not something strict; you can play around with this if you like.)

Example:

```
Sudoku> sample cell
Just 3
Nothing
Nothing
Just 7
Nothing
```

C2. Make Sudokus an instance of the class Arbitrary.

```
instance Arbitrary Sudoku where
  ...
```

We have already done this for you in the file **Sudoku.hs**.

C3. Define a property

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

that expresses that each generated Sudoku actually is a Sudoku according to Assignment A2. Also use

QuickCheck to check that the property actually holds for all Sudokus that are generated.

Hints

Here are some functions that might come in handy:

```
sample    :: Show a => Gen a -> IO ()
choose    :: Random a => (a,a) -> Gen a
frequency :: [(Int,Gen a)] -> Gen a
sequence  :: [Gen a] -> Gen [a]
```

You might want to take a look at the lecture notes and example code on test data generation.

Rows, Columns, Blocks

Now, we are going to think about what actually constitutes a valid solution of a Sudoku. There are three constraints that a valid solution has to forfill:

- No row can contain the same digit twice;
- No column can contain the same digit twice;
- No 3x3 block can contain the same digit twice.

This leads us to the definition of a **block**; a block is either a row or a column or a 3x3 block. A block therefore contains 9 cells:

```
type Block = [Maybe Int]
```

We are going to define a function that checks if a Sudoku is not violating any of the above constraints, by checking that none of the blocks violate those constraints.

Assignment D

D1. Implement a function:

```
isOkayBlock :: Block -> Bool
```

that, given a block, checks if that block does not contain the same digit twice.

Examples:

```
Sudoku> isOkayBlock [Just 1, Just 7, Nothing, Nothing, Just 3, Nothing, Nothing, Nothing, Just 2]
True
Sudoku> isOkayBlock [Just 1, Just 7, Nothing, Just 7, Just 3, Nothing, Nothing, Nothing, Just 2]
False
```

D2. Implement a function:

```
blocks :: Sudoku -> [Block]
```

that, given a Sudoku, creates a list of all blocks of that Sudoku. This means:

- 9 rows,
- 9 columns,
- 9 3x3 blocks.

Also add a property that states that, for each Sudoku, there are 3*9 blocks, and each block has exactly 9 cells.

D3. Now, implement a function:

```
isOkay :: Sudoku -> Bool
```

that, given a Soduko, checks that all rows, colums and 3x3 blocks do not contain the same digit twice.

Examples:

```
Sudoku> isOkay allBlankSudoku
True
Sudoku> do sud <- readSudoku "example.sud"; print (isOkay sud)
True
```

Hints

Here are some functions that might come in handy:

```
nub       :: Eq a => [a] -> [a]
transpose :: [[a]] -> [[a]]
```

```
take      :: Int -> [a] -> [a]
drop      :: Int -> [a] -> [a]
```

Note that some of the above functions only appear when you import **Data.List**.

You might want to take a look at the exercises and answers on lists and list comprehensions.

Positions and Finding Blanks

We are getting closer to the final solving function. Let us start thinking about how such a function would work.

Given a Sudoku, if there are no blanks left in the Sudoku, we are done. Otherwise, there is at least one blank cell that needs to be filled in somehow. We are going to write functions to find and manipulate blank cells.

It is quite natural to start to talk about **positions**. A position is a coordinate that identifies a cell in the Sudoku. Here is a way of modelling coordinates:

```
type Pos = (Int,Int)
```

We use positions as indicating first the row and then the column. For example, the position (3,5) denotes the 5th cell in the 3rd row.

Note: It is common in programming languages to start counting at 0! Therefore, the position that indicates the upper left corner is (0,0), and the position indicating the lower right corner is (8,8).

Assignment E

E1. Implement a function:

```
blanks :: Sudoku -> [Pos]
```

that, given a Sudoku returns a list of the positions in the Sudoku that are still blank. You may decide on the order in which the positions appear.

Examples:

```
Sudoku> length (blanks allBlankSudoku) == 9*9
True
Sudoku> blanks example
[(0,2),(0,3),(0,7),(0,8),(1,0),(1,2),(1,3),(1,4),(1,5),(1,8),(2,0),(2,1),
 (2,4),(2,7),(2,8),(3,0),(3,1),(3,2),(3,3),(3,6),(4,1),(4,2),(4,4),(4,6),
 (4,7),(5,2),(5,5),(5,6),(5,7),(5,8),(6,0),(6,1),(6,4),(6,7),(6,8),(7,0),
 (7,3),(7,4),(7,5),(7,6),(7,8),(8,0),(8,1),(8,5),(8,6)]
```

In addition, write a property that states that all cells in the blanks list are actually blank.

E2. Implement a function:

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

that, given a list, and a tuple containing an index in the list and a new value, updates the given list with the new value at the given index.

Examples:

```
Sudoku> ["a","b","c","d"] !!= (1,"apa")
["a","apa","c","d"]
Sudoku> ["p","qq","rrr"] !!= (0,"bepa")
["bepa","qq","rrr"]
```

Also write (a) propert(y/ies) that state(s) the expected properties of this function. Think about what can go wrong!

E3. Implement a function:

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

that, given a Sudoku, a position, and a new cell value, updates the given Sudoku at the given position with the new value.

Example:

```
Sudoku> printSudoku (update allBlankSudoku (1,3) (Just 5))
.........
...5.....
.........
.........
```

```
.........
.........
.........
.........
.........
```

Also write a property that checks that the updated position really has gotten the new value.

E4. Implement a function:

```
candidates :: Sudoku -> Pos -> [Int]
```

that, given a Sudoku, and a blank position, determines which numbers could be legally written into that position.

Example:

```
Sudoku> candidates example (0,2)
[4,8]
Sudoku> candidates allBlankSudoku (8,8)
[1,2,3,4,5,6,7,8,9]
```

In addition, write a property that relates the function candidates with the functions update, isSudoku, and isOkay. (This property can be very useful to understand how to solve Sudokus!)

Hints

There is a standard function **(!!)** in Haskell for getting a specific element from a list. It starts indexing at 0, so for example to get the first element from a list xs, you can use xs !! 0.

We usually use the standard function **zip** to pair up elements in a list with their corresponding index.

Example:

```
Prelude> ["apa","bepa","cepa"] `zip` [1..3]
[("apa",1),("bepa",2),("cepa",3)]
```

This, in combination with list comprehensions, should be very useful for this assignment!

When testing a property that is polymorphic (meaning that it has type variables in its type), you need to add a type signature that picks an arbitrary type. For example, when testing properties for the function (!!=), which works for lists of any type, you have to fix the type when testing, for example lists of Integers. Do this by adding a type signature to your properties.

Here are some more useful functions:

```
head :: [a] -> a
(!!) :: [a] -> Int -> a
zip  :: [a] -> [b] -> [(a,b)]
```

Solving Sudokus

Finally, we have all the bits in place to attack our main problem: Solving a given Sudoku.

Our objective is to define a Haskell function

```
solve :: Sudoku -> Maybe Sudoku
```

The basic idea is as follows. Function solve must first check that its argument is not already a bad Sudoku. This means that (1) it represents a 9x9 sudoku, (2) it has no blocks (rows, columns, 3x3 blocks) that contain the same digit twice. We will only do this check once. If the argument is bad then solve must return **Nothing**

Now if we have such a Sudoku **sud** that we would like to solve, we give it to a recursive helper function **solve'**.

The **solve'** function must consider all the blanks in **sud**. If this list is empty then by (1) and (2) above we are done, and the answer of **solve'** (and hence **solve**) must be **Just sud**.

Otherwise there is at least one blank position. We choose one of them. For this blank position we we try to **recursively** solve **sud**, once for each possible candidate; in each recursive case we update the blank cell with a candidate. The first recursive attempt that does not give **Nothing** provides our solution. But if none of the recursive attempts succeed, we return **Nothing**. This method of problem solving is called **backtracking**.

Assignment F

F1. Implement a function:

```
solve :: Sudoku -> Maybe Sudoku
```

using the above idea.

Examples:

```
Sudoku> printSudoku (fromJust (solve allBlankSudoku))
123456789
456789123
789123456
214365897
365897214
897214365
531642978
642978531
978531642
Sudoku> do sud <- readSudoku "example.sud"; printSudoku (fromJust (solve sud))
364871295
752936184
819254736
596713428
431582679
278469351
645328917
983147562
127695843
Sudoku> do sud <- readSudoku "impossible.sud"; print (solve sud)
Nothing
```

(In the above examples, we use the standard function **fromJust** from the library **Data.Maybe**.)

F2. For your own convenience, define a function:

```
readAndSolve :: FilePath -> IO ()
```

that produces instructions for reading the Sudoku from the given file, solving it, and printing the answer.

Examples:

```
Sudoku> readAndSolve "example.sud"
364871295
752936184
819254736
596713428
431582679
278469351
645328917
983147562
127695843
Sudoku> readAndSolve "impossible.sud"
(no solution)
```

F3. Implement a function:

```
isSolutionOf :: Sudoku -> Sudoku -> Bool
```

that checks, given two Sudokus, whether the first one is a solution (i.e. all blocks are okay, there are no blanks), and also whether the first one is a solution of the second one (i.e. all digits in the second sudoku are maintained in the first one).

Examples:

```
Sudoku> fromJust (solve allBlanksSudoku) `isSolutionOf` allBlanksSudoku
True
Sudoku> allBlankSudoku `isSolutionOf` allBlanksSudoku
False
Sudoku> fromJust (solve allBlankSudoku) `isSolutionOf` example
False
```

F4. Define a property:

```
prop_SolveSound :: Sudoku -> Property
```

that says that the function **solve** is **sound**. Soundness means that every supposed solution produced by **solve** actually is a valid solution of the original problem.

Hints

All the work we did in the assignments A -- E should be used in order to implement the function **solve**.

QuickChecking the property **prop_SolveSound** will probably take a long time. Be patient! Alternatively, there are a number of things you can do about this.

- You can test on fewer examples (using the QuickCheck function quickCheckWith). You can for example define:
  ```
  fewerChecks prop = quickCheckWith stdArgs{ maxSuccess = 30 } prop
  ```
  and then write **fewerChecks prop_SolveSound** when you want to QuickCheck the property.
- You can also generate Sudokus with a different probability distribution. Try increasing the amount of digits in an arbitrary Sudoku by fiddling with the frequencies in the **cell** function from Assignment C1 and see what happens.
- You can use a compiler, such as **GHC**.

It is okay if you do not find a completely satisfactory solution to this issue.

Here are some useful functions:

```
fromJust    :: Maybe a -> a
listToMaybe :: [a] -> Maybe a
catMaybes   :: [Maybe a] -> [a]
```

Here is an example of an impossible Sudoku:

- **impossible.sud**

Extra Assignments (for fun)

Just for fun. You can choose freely whether to do 0, 1 or more of these. Don't expect us to spend time grading these however. There are no perfect, pre-defined answers here.

- **Bring it on!**

Submission

Submit your solutions using the Fire system.

Your submission should consist of the following file:

- **Sudoku.hs**, containing your solution. It should contain enough comments to understand what is going on.

Before you submit your code, Clean It Up! Remember, submitting clean code is Really Important, and simply the polite thing to do. After you feel you are done, spend some time on cleaning your code; make it simpler, remove unneccessary things, etc. We will reject your solution if it is not clean. Clean code:

- Does not have long lines (< 78 characters)
- Has a consistent layout
- Has type signatures for all top-level functions
- Has good comments
- Has no junk (junk is unused code, commented code, unneccessary comments)
- Has no overly complicated function definitions
- Does not contain any repetitive code (copy-and-paste programming)

To the Fire System Good Luck!

Lab written and developed by **Koen Lindström Claessen**