Universiteit van Amsterdam

Opgave 2

Erik Kooistra en Ana Oprescu

Sudoku

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

Deadline: Individueel: woensdag 13 februari 2019, 23:59
Team: zondag 17 februari 2019, 23:59

1 Introductie en puzzeltjes (Individueel)

Voordat je begint met het grotere Haskell-programma dat je deze week gaat schrijven, zijn er wat kleinere puzzeltjes die je kunt oplossen. Dit vragen we niet om je te pesten en ook niet om te kijken hoe goed je stukken code op het internet kunt zoeken, maar omdat je hierdoor wat handigheid met en begrip over Haskell zult ontwikkelen. Op het hoorcollege is behandeld dat er in Haskell enkele standaardfuncties zitten die de taal krachtiger maken en nu ga je deze functies ook echt toepassen. Hieronder staan tien functies die met behulp van deze standaardfuncties gedefinieerd kunnen worden, doe dit en gebruik je opgedane kennis bij de rest van de opdracht.

1. Definieer `length` in termen van `foldr`.
2. Kom erachter wat `or` doet en definieer je eigen versie in termen van `foldr`.
3. Definieer `elem x` in termen van `foldr`[1].
4. Definieer `map f` in termen van `foldr`.
5. Definieer `(++)` in termen van `foldr`.
6. Definieer `reverse` in termen van `foldr`.
7. Definieer `reverse` in termen van `foldl`.
8. (expert) Definieer `(!!)` in termen van `foldl`.
9. Maak een functie `isPalindrome` die bepaalt of een string (of lijst) een palindroom is.
10. Maak een functie die een oneindige lijst met de Fibonacci reeks teruggeeft in termen van `scanl`.
11. Gegeven de volgende type definitie `type List = (Int) -> Int`, schrijf een functie die het mogelijk maakt op elementen toe te voegen aan dit type

Lever deze functies in als de module `Puzzles`, door een bestand "`Puzzles.hs`" aan te maken met bovenaan:

```
module Puzzles

where
```

gevolgd door jouw uitwerkingen (in de juiste volgorde).

[1] Om dit te laten werken moet je Haskell hinten dat de parameter in de Eq-typeclass zit, bijvoorbeeld:
`elem' :: Eq a => a -> [a] -> Bool`

2 Sudoku Team

De Sudoku is een algemeen bekende puzzel; achtergrondinformatie en regels erover zijn makkelijk online te vinden. De bedoeling is dat je een algemene Sudoku solver implementeert in het bestand `SudokuSolver.hs`.

```
module SudokuSolver

where

import Data.List

type Row    = Int
type Column = Int
type Value  = Int
type Grid   = [[Value]]

positions, values :: [Int]
positions = [1..9]
values    = [1..9]

blocks :: [[Int]]
blocks = [[1..3],[4..6],[7..9]]

nrcBlocks :: [[Int]]
nrcBlocks = [[2..4],[6..8]]

showDgt :: Value -> String
showDgt 0 = " "
showDgt d = show d

showRow :: [Value] -> IO()
showRow [a1,a2,a3,a4,a5,a6,a7,a8,a9] =
 do  putChar '|'         ; putChar ' '
     putStr (showDgt a1) ; putChar ' '
     putStr (showDgt a2) ; putChar ' '
     putStr (showDgt a3) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a4) ; putChar ' '
     putStr (showDgt a5) ; putChar ' '
     putStr (showDgt a6) ; putChar ' '
     putChar '|'         ; putChar ' '
     putStr (showDgt a7) ; putChar ' '
     putStr (showDgt a8) ; putChar ' '
     putStr (showDgt a9) ; putChar ' '
     putChar '|'         ; putChar '\n'

showGrid :: Grid -> IO()
showGrid [as,bs,cs,ds,es,fs,gs,hs,is] =
 do putStrLn ("+-------+-------+-------+")
    showRow as; showRow bs; showRow cs
    putStrLn ("+-------+-------+-------+")
    showRow ds; showRow es; showRow fs
    putStrLn ("+-------+-------+-------+")
    showRow gs; showRow hs; showRow is
```

```
    putStrLn ("+-------+-------+-------+")

type Sudoku = (Row,Column) -> Value

sud2grid :: Sudoku -> Grid
sud2grid s =
  [ [ s (r,c) | c <- [1..9] ] | r <- [1..9] ]

grid2sud :: Grid -> Sudoku
grid2sud gr = \ (r,c) -> pos gr (r,c)
  where
  pos :: [[a]] -> (Row,Column) -> a
  pos gr (r,c) = (gr !! (r-1)) !! (c-1)

showSudoku :: Sudoku -> IO()
showSudoku = showGrid . sud2grid
```

Het doel van de solver is een sudoku als een Grid accepteren en een opgeloste sudoku, als een Grid, teruggeven. Een voorbeeld van zo'n Grid is als volgt.

```
example1 :: Grid
example1 = [[5,3,0,0,7,0,0,0,0],
            [6,0,0,1,9,5,0,0,0],
            [0,9,8,0,0,0,0,6,0],
            [8,0,0,0,6,0,0,0,3],
            [4,0,0,8,0,3,0,0,1],
            [7,0,0,0,2,0,0,0,6],
            [0,6,0,0,0,0,2,8,0],
            [0,0,0,4,1,9,0,0,5],
            [0,0,0,0,8,0,0,7,9]]
```

De eerste stap is het bepalen welke mogelijkheden er zijn gegeven een rij (row), kolom (column) en een blok (subgrid). Daarnaast hebben we nog een lijst nodig van alle lege posities in de sudoku. Hiervoor zijn de volgende functies nodig.

1. `extend :: Sudoku -> (Row,Column,Value) -> Sudoku`
2. `freeInRow :: Sudoku -> Row -> [Value]`
3. `freeInColumn :: Sudoku -> Column -> [Value]`
4. `freeInSubgrid :: Sudoku -> (Row,Column) -> [Value]`
5. `freeAtPos :: Sudoku -> (Row,Column) -> [Value]`
6. `openPositions :: Sudoku -> [(Row,Column)]`

Nu we kunnen uitrekenen wat de mogelijkheden zijn voor elke vrije plek in de sudoku moeten we ook kunnen controleren of de sudoku geldig is voor de gegeven oplossing.

1. `rowValid :: Sudoku -> Row -> Bool`
2. `colValid :: Sudoku -> Column -> Bool`
3. `subgridValid :: Sudoku -> (Row,Column) -> Bool`
4. `consistent :: Sudoku -> Bool`

2.1 Sudoku oplossen

Met al deze verschillende functies hebben we nu voldoende informatie om te gaan beginnen aan het oplossen van een sudoku, dit gaan we doen door middel van een search tree. Maar eerst moeten we daarvoor nog wat types definiëren. Een Constraint heeft als waarde een x,y coördinaat in de sudoku en een lijst met mogelijke opties die daar kunnen.

```
type Constraint = (Row,Column,[Value])
type Node = (Sudoku,[Constraint])

showNode :: Node -> IO() showNode = showSudoku . fst
```

Als eerste stap in het oplossen, is een lijst genereren van alle mogelijke Constraints gesorteerd van minste opties naar meeste opties. De functie definitie daarvoor is als volgt `constraints :: Sudoku -> [Constraint]`

Met al deze informatie is het nu mogelijk om de solve functie te schrijven. Deze heeft als definitie `solveAndShow :: Grid -> IO[()]`

Hoe je dit het beste kan implementeren is als eerste een lijst van initial constraints te maken, en dan per mogelijkheid verder zoeken naar nieuwe mogelijkheden tot je niet meer verder kan.