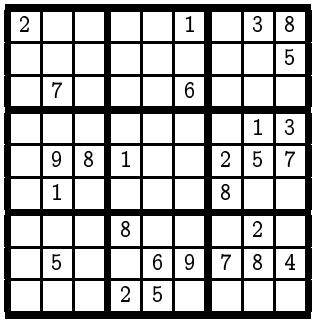
7 Sudoku (fit the first)

Standard Sudoku puzzles consist of a 9×9 grid of squares, some of which are blank and the others which contain a single digit drawn from 1 to 9.



```
["2....1.38",
 ".........5",
 ".7...6...",
 ".......13",
 ".981..257",
 "31....8..",
 "9..8...2.",
 ".5..69784",
 "4..25...."]
```

The solution is the unique square obtained by filling the empty squares with single digits so that each row, each column, and each of the non-overlapping 3×3 blocks contains all digits exactly once.

We don't deign to solve the puzzle, what we do is write a function that computes all possible completions of a puzzle.

```
> type Solver = Grid -> [ Grid ]
```

7.1 Specifying the problem

A grid will be a matrix of digits. The contents of a cell in the grid will be a digit or a blank: we might use a data-type with ten values and have the type checker make sure that we always have valid contents in a cell, but it will be easier not to. We assume the input contains only the ten valid characters. For pragmatic reasons, blanks are represented by a visible character. Similarly we choose to represent a matrix as a list of rows, and a row as a list of columns, so that we can use familiar list operations.

```
> type Grid = Matrix Digit
> type Digit = Char

> type Matrix a = [ Row a ]
> type Row a    = [ Cell a ]
> type Cell a   = a

> digits :: [ Digit ]
> digits = ['1'..'9']

> blank :: Digit -> Bool
> blank = (== '.')
```

Perhaps the most straightforward way of defining a solver

```
> solve :: Solver
> solve = filter legal . expand . freedoms
```

is to list all possible choices available for each cell, consider all possible grids that can be made to match those possibilities, and filter for the one(s) that are legal solutions.

```
freedoms :: Grid -> Matrix Freedoms
expand   :: Matrix Freedoms -> [ Grid ]
legal    :: Grid -> Bool
```

We can represent the freedom for each cell as a list of possible (genuine) digits

```
> type Freedoms = [ Digit ]
```

```
> freedoms :: Grid -> Matrix Freedoms
> freedoms = map ( map choice )
>            where choice d | blank d   = digits
>                           | otherwise = [d]
```

Expanding a matrix of such lists is the Cartesian product for matrices.

```
> expand :: Matrix Freedoms -> [ Grid ]
> expand = cp . map cp
```

where the Cartesian product for lists

```
> cp :: [[a]] -> [[a]]
> cp     []   = [[]]
> cp (xs:xss) = [ x:ys | x <- xs, ys <- cp xss ]
```

lists all possible ways of choosing exactly one value from each list in a list of lists. For example, $cp\ [[1,2],[3],[4,5]] = [[1,3,4],[1,3,5],[2,3,4],[2,3,5]]$.

(Oh, look, *cp* is a fold.)

Finally, an acceptable grid is one in which no row, column or box contains duplicates:

```
> legal :: Grid -> Bool
> legal g = all nodups (rows g) &&
>           all nodups (cols g) &&
>           all nodups (boxs g)
```

The most direct implementation of *nodups* is

```
> nodups :: Eq a => [a] -> Bool
> nodups    []  = True
> nodups (x:xs) = x `notElem` xs && nodups xs
```

where the function

```
notElem x xs = all (/= x) xs
```

is predefined.

This test takes $O(n^2)$ steps on a list of length n. We might choose to sort the list and check that it is strictly increasing, and sorting can be done in $O(n \log n)$ steps so this is $O(n \log n)$. However for small problems, here $n = 9$, it is not clear that using an efficient sorting algorithm is worthwhile because of the constant of proportionality. (Would you prefer $9n^2$ steps or $30n \log_2 n$ steps?)

7.2 Rows, columns and box(e)s

Since a matrix is given by a list of its rows, the function *rows* is just the identity function on matrices:

```
> rows :: Matrix a -> [ Row a ]
> rows = id
```

and the function *cols* computes the transpose of a matrix:

```
> cols :: Matrix a -> [ Row a ]
> cols    [xs]  = [ [x] | x <- xs ]
> cols (xs:xss) = zipWith (:) xs (cols xss)
```

The predefined function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

zips together a pair of lists, like *zip*, but by applying its function argument to each pair of corresponding elements to produce the corresponding result.

(Oh, look, *cols* is almost a fold... in fact it is a fold on non-empty lists, but by extending it to empty lists it can be made to be an instance of *fold*.)

Dividing a list into groups of n can be done by

```
> by :: Int -> [a] -> [[a]]
> by n [] = []
> by n xs = take n xs : by n (drop n xs)
```

(which is an unfold, known as *chunk* in some libraries).

The function that identifies the squares that are ninths of the grid

```
> boxs :: Matrix a -> [ Row a ]
> boxs = map concat . concat . map cols . by 3 . map (by 3)
```

divides each row into threes, parts the rows into threes, transposes the small matrices in each part, and then joins the whole back into a single list of the boxes.

This might be better illustrated by a similar transformation of a smaller 4×4 matrix into quarters:

```
[[a,b,c,d],       [[[a,b],[c,d]],      [[[[a,b],[c,d]],
 [e,f,g,h],  ⟶     [[e,f],[g,h]],  ⟶     [[e,f],[g,h]]]   ⟶
 [i,j,k,l],        [[i,j],[k,l]],       [[[i,j],[k,l]],
 [m,n,o,p]]        [[m,n],[o,p]]]        [[m,n],[o,p]]]]

  [[[[a,b],[e,f]],        [[[a,b],[e,f]],       [[a,b,e,f],
    [[c,d],[g,h]]]  ⟶      [[c,d],[g,h]],  ⟶     [c,d,g,h],
   [[[i,j],[m,n]],         [[i,j],[m,n]],       [i,j,m,n],
    [[k,l],[o,p]]]]        [[k,l],[o,p]]]       [k,l,o,p]]
```

One of the many virtues of functional programming is that it encourages this holistic style of programming, which manages whole objects rather than fiddling with many subscripts.

7.3 Infeasibility

This completes the implementation of

```
> solve :: Solver
> solve = filter legal . expand . freedoms
```

This implements exactly the function that maps a Sudoku problem to its solution. It is however entirely useless for solving Sudoku problems.

The uniqueness of the solution can be enforced by as few as 17 non-blank squares in the puzzle, so there can be $9^{9\times 9-17} \approx 10^{61}$ potential solutions to be checked. There are about 3×10^7 seconds in a year, so checking 10^{61} grids at the rate of a million a second would take of the order of 3×10^{47} years, which compares unfavourably with estimates of the age of the universe of the order of 10^{10} years.

Even a relatively easy problem such that given earlier in the notes will have many fewer than half of the squares filled in. No technological improvement is going to help here, nor is any realistic amount of concurrent computation.

This algorithm is not *slow*, it is *infeasible*.

7.4 Holistic reasoning

Thinking about whole data structures makes it easier to reason about programs, helped by the *point-free* (or *tacit*) style which encourages the omission of unnecessary arguments, so

```
> solve = filter legal . expand . freedoms
```

rather than

```
> solve xss = filter legal (expand (freedoms xss))
```

Not only are there no subscripts, nor individual elements of the grids, mentioned in the program. Point-free style here omits all mention of the grid itself, except perhaps in the type.

For example, the grid arranging functions are all their own inverses

$$rows \cdot rows = cols \cdot cols = boxs \cdot boxs = id$$

at least on 9×9 matrices. Moreover,

$$\begin{aligned} map\ rows \cdot expand &= expand \cdot rows \\ map\ cols \cdot expand &= expand \cdot cols \\ map\ boxs \cdot expand &= expand \cdot boxs \end{aligned}$$

on all 9×9 matrices of choices. There are also a wealth of laws about standard functions like

$$\begin{aligned} map\ f \cdot concat &= concat \cdot map\ (map\ f) \\ filter\ p \cdot concat &= concat \cdot map\ (filter\ p) \\ map\ f \cdot filter\ (p \cdot f) &= filter\ p \cdot map\ f \\ filter\ (all\ p) \cdot cp &= cp \cdot map\ (filter\ p) \end{aligned}$$

Exercises

7.1 Express the Cartesian product function

```
> cp :: [[a]] -> [[a]]
> cp []       = [[]]
> cp (xs:xss) = [ x:ys | x <- xs, ys <- cp xss ]
```

from the lectures as an instance of *fold* (or the standard function *foldr*).

7.2 The function

```
> cols :: [[a]] -> [[a]]
> cols    [xs]  = [ [x] | x <- xs ]
> cols (xs:xss) = zipWith (:) xs (cols xss)
```

is not quite in the form of a fold (on lists) because there is a special case for singelton lists. Define a function *cols′* which agrees with *cols* wherever that function is defined, and for which *cols′* [] has a value that makes the equation for *cols* [*xs*] redundant. This should give the definition of *cols′* the form of a fold. Finally, write *cols′* as an instance of *fold*.

7.3 A type of non-empty lists with elements of type a might have been defined by

```
> data Pist a = Wrap a | Pons a (Pist a)
```

Recall that the fold function for lists

```
> fold :: (a -> b -> b) -> b -> [a] -> b
```

can be characterised by *fold* (:) [] being the identity function on lists. The corresponding property for the fold on non-empty lists

```
>  foldp :: ... -> ... -> Pist a -> b
```

is that *foldp Pons Wrap* is the identity on *Pist a*.

Complete the definition of *foldp*, and write functions

```
pist :: [a] -> Pist a
list :: Pist a -> [a]
```

which embed *Pist a* in [*a*], so that for example

$$pist\ [1,2,3] = Pons\ 1\ (Pons\ 2\ (Wrap\ 3))$$

and

$$list\ (Pons\ 1\ (Pons\ 2\ (Wrap\ 3))) = [1,2,3]$$

What can you say about $list \cdot pist$ and $pist \cdot list$?

Let $fold1\ cons\ wrap = foldp\ cons\ wrap \cdot pist$. Calculate, or write out, a direct recursive definition of *fold1*.

Show that *cols* can be expressed as an instance of *fold1*.

There is a predefined function

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f   [x]  = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Can you write *foldr1 f* as an instance of *fold1*?

Can you write *fold1 cons wrap* as an instance of *foldr1*?