IN3043 Functional Programming
Solutions to Exercises 6

1. This is easily done by converting a list to a set and back again:

```
unique :: Ord a => [a] -> [a]
unique xs = Set.elems (Set.fromList xs)
```

   We could also write this using function composition:

```
unique :: Ord a => [a] -> [a]
unique = Set.elems . Set.fromList
```

2. We can use **`Map.unionsWith`** to add counts, so we start by mapping each element of the list into a singleton map with count 1, and add those.

```
frequencyMap :: Ord a => [a] -> Map a Int
frequencyMap xs =
    Map.unionsWith (+) [Map.singleton x 1 | x <- xs]
```

   Again, we could also write this using function composition:

```
frequencyMap :: Ord a => [a] -> Map a Int
frequencyMap =
    Map.unionsWith (+) . map (\ x -> Map.singleton x 1)
```

3. (a) Since we have **`bfs`**, this is trivial:

```
furtherest :: Maze -> Set Point
furtherest (open, start, finish) =
    last (bfs (moves open) start)
```

   (b) This is similar to **`solve`**, except that we want the union of all those sets:

```
nearer :: Maze -> Set Point
nearer (open, start, finish) =
    Set.unions $
    takeWhile (\ s -> not (Set.member finish s)) $
    bfs (moves open) start
```

   (c) We can use **`dropWhile`** to discard the nearer nodes. The first remaining set (if any), will be the nodes at the same distance as the goal. We can discard those with **`drop 1`**. (Unlike **`tail`**, this works even if the list is empty.) Then we want the union of the remaining sets.

```
further :: Maze -> Set Point
further (open, start, finish) =
    Set.unions $
    drop 1 $
    dropWhile (\ s -> not (Set.member finish s)) $
    bfs (moves open) start
```

4. (a) This will be a record containing four binary values, which could be represented in several ways. We choose a **data** type with four **Bool** entries for the farmer, wolf, goat and cabbage respectively, with **False** indicating the near bank and **True** indicating the far bank:

```
-- crossed the river? False = near bank, True = far
   bank
type Bank = Bool

-- crossed: farmer, wolf, goat, cabbage
data State = State Bank Bank Bank Bank
    deriving (Eq, Ord, Show)
```

(b) At the start, all four are on the near bank:

```
start :: State
start = State False False False False
```

The goal is a state where all four are on the far bank:

```
finish :: State
finish = State True True True True
```

(c) If the goat is on the same bank as the farmer, nothing will happen. Otherwise, the goat must be away from the wolf and the cabbage:

```
safe :: State -> Bool
safe (State f w g c) = g == f || g /= w && g /= c
```

(There are several equivalent ways of defining this.)

(d) Because we have chosen to represent a bank with **Bool**, the opposite bank can be obtained with **not**:

```
cross :: Bank -> Bank
cross = not
```

The farmer can either cross alone or take one of the items on the same bank as him. We then need to select resulting states that are safe.

```
crossings :: State -> Set State
crossings (State f w g c) =
    Set.fromList $ filter safe $
        [State (cross f) w g c] ++
        [State (cross f) (cross w) g c | w == f] ++
        [State (cross f) w (cross g) c | g == f] ++
        [State (cross f) w g (cross c) | c == f]
```

(e) This is similar to the maze solver:

interpreter

```
length $ takeWhile (\ s -> not (Set.member finish s))
   $ bfs crossings start
```

5. (a) This function uses **readGrid** in a similar manner to **readMaze**:

```
readLife :: String -> LiveCells
readLife s =
    Set.fromList [p | (p, c) <- readGrid s, c == '@']
```

(b) There are two cases:

```
showCell :: LiveCells -> Point -> Char
showCell ps p
  | Set.member p ps = '@'
  | otherwise = '.'
```

(c) We have to reverse the y-coordinates, because larger ones should come first:

```
showGrid :: Point -> Point -> (Point -> Char) -> String
showGrid (minx, miny) (maxx, maxy) cell =
    unlines [[cell (x, y) | x <- [minx..miny]],
        y <- reverse [miny..maxy]]
```

(d) We do this in two stages:

```
neighbours :: Point -> Set Point
neighbours p = Set.fromList [plusPoint p d | d <-
   offsets]

offsets :: [Point]
offsets =
    [Point dx dy | dx <- [-1..1], dy <- [-1..1],
        dx /= 0 || dy /= 0]
```

(e) This is a direct translation of Conway's rules:

```
rule :: Bool -> Int -> Bool
rule True n = n == 2 || n == 3
rule False n = n == 3
```

(f) The set of live neighbours of a point is the intersection of the set of neighbours with the set of live points.

```
live :: LiveCells -> Point -> Bool
live ps p =
    rule (Set.member p ps) (Set.size (Set.intersection
       (neighbours p) ps))
```

(g) We combine the neighbours of all live cells with the live cells to get the candidates for the next generation, and then filter them using Conway's rule to get the next generation.

```
generation :: LiveCells -> LiveCells
generation ps =
    Set.filter (live ps) $
    Set.union ps (Set.unions (map neighbours
       (Set.elems ps)))
```

Alternatively we could have written

```
generation :: LiveCells -> LiveCells
generation ps =
    Set.filter (live ps) $ Set.union ps $ Set.unions $
    map neighbours $ Set.elems ps
```