CS 381, Spring 2020 Homework 1 (Haskell)

Please follow carefully ***all*** of the following steps:

1. Prepare a Haskell (or literate Haskell) file (ending in `.hs` or `.lhs`, respectively) that compiles without errors in GHCi. (Put all non-working parts in comments.)
2. Submit ***one*** solution per group (each group can have up to 5 members) through Canvas.

Late submissions will ***not*** be accepted. Do ***not*** send solutions by email.

Insert the following definitions at the top of your file.

```
import Data.List (nub,sort)

norm :: Ord a => [a] -> [a]
norm = sort . nub
```

Exercise 1. Programming with Lists

Multisets, or bags, can be represented as list of pairs (x, n) where n indicates the number of occurrences of x in the multiset.

```
type Bag a = [(a,Int)]
```

For the following exercises you can assume the following properties of the bag representation. *But note:* Your function definitions have to maintain these properties for any multiset they produce!

(1) Each element x occurs in at most one pair in the list.
(2) Each element that occurs in a pair has a positive counter.

As an example consider the multiset $\{2, 3, 3, 5, 7, 7, 7, 8\}$, which has the following representation (among others).

```
[(5,1),(7,3),(2,1),(3,2),(8,1)]
```

Note that the order of elements is not fixed. In particular, we cannot assume that the elements are sorted. Thus, the above list representation is just one example of several possible.

(a) Define the function `ins` that inserts an element into a multiset.

```
ins :: Eq a => a -> Bag a -> Bag a
```

(*Note:* The class constraint "`Eq a =>`" restricts the element type `a` to those types that allow the comparison of elements for equality with ==.)

(b) Define the function `del` that removes an element from a multiset.

```
del :: Eq a => a -> Bag a -> Bag a
```

(c) Define a function `bag` that takes a list of values and produces a multiset representation.

```
bag :: Eq a => [a] -> Bag a
```

For example, with `xs = [7,3,8,7,3,2,7,5]` we get the following result.

```
> bag xs
[(5,1),(7,3),(2,1),(3,2),(8,1)]
```

(*Note:* It's a good idea to use of the function `ins` defined earlier.)

(d) Define a function `subbag` that determines whether or not its first argument bag is contained in the second.

```
subbag :: Eq a => Bag a -> Bag a -> Bool
```

Note that a bag b is contained in a bag b' if every element that occurs n times in b occurs also at least n times in b'.

(e) Define a function `isbag` that computes the intersection of two multisets.

```
isbag :: Eq a => Bag a -> Bag a -> Bag a
```

(f) Define a function `size` that computes the number of elements contained in a bag.

```
size :: Bag a -> Int
```

Exercise 2. Graphs

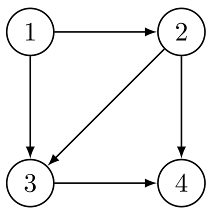
A simple way to represent a directed graph is through a list of edges. An edge is given by a pair of nodes. For simplicity, nodes are represented by integers.

```
type Node  = Int
type Edge  = (Node,Node)
type Graph = [Edge]
type Path  = [Node]
```

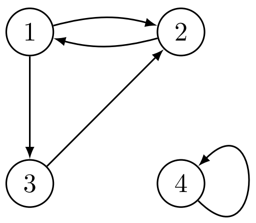
(We ignore the fact that this representation cannot distinguish between isolated nodes with and without loops; see, for example, the loop/edge `(4,4)` in the graph h that represents an isolated node.)

Consider, for example, the following directed graphs.

$g =$



$h =$



These two graphs are represented as follows.

```
g :: Graph
g = [(1,2),(1,3),(2,3),(2,4),(3,4)]

h :: Graph
h = [(1,2),(1,3),(2,1),(3,2),(4,4)]
```

Note: In some of your function definitions you might want to use the function `norm` to remove duplicates from a list and sort it.

(a) Define the function `nodes :: Graph -> [Node]` that computes the list of nodes contained in a given graph. For example, `nodes g = [1,2,3,4]`.

(b) Define the function `suc :: Node -> Graph -> [Node]` that computes the list of successors for a node in a given graph. For example, `suc 2 g = [3,4]`, `suc 4 g = []`, and `suc 4 h = [4]`.

(c) Define the function `detach :: Node -> Graph -> Graph` that removes a node together with all of its incident edges from a graph. For example, `detach 3 g = [(1,2),(2,4)]` and `detach 2 h = [(1,3),(4,4)]`.

(d) Define the function `cyc :: Int -> Graph` that creates a cycle of any given number. For example, `cyc 4 = [(1,2),(2,3),(3,4),(4,1)]`.

Note: All functions can be succinctly implemented with list comprehensions.

Exercise 3. Programming with Data Types

Here is the definition of a data type for representing a few basic shapes. A figure is a collection of shapes. The type `BBox` represents *bounding boxes* of objects by the points of the lower-left and upper-right hand corners of the smallest enclosing rectangle.

```
type Number = Int

type Point = (Number,Number)
type Length = Number

data Shape = Pt Point
           | Circle Point Length
           | Rect Point Length Length
           deriving Show

type Figure = [Shape]

type BBox = (Point,Point)
```

(a) Define the function `width` that computes the width of a shape.

```
width :: Shape -> Length
```

For example, the widths of the shapes in the figure `f` are as follows.

```
f = [Pt (4,4), Circle (5,5) 3, Rect (3,3) 7 2]

> map width f
[0,6,7]
```

(b) Define the function `bbox` that computes the bounding box of a shape.

```
bbox  :: Shape -> BBox
```

The bounding boxes of the shapes in the figure `f` are as follows.

```
> map bbox f
[((4,4),(4,4)),((2,2),(8,8)),((3,3),(10,5))]
```

(c) Define the function `minX` that computes the minimum *x* coordinate of a shape.

```
minX  :: Shape -> Number
```

The minimum *x* coordinates of the shapes in the figure `f` are as follows.

```
> map minX f
[4,2,3]
```

(d) Define a function `move` that moves the position of a shape by a vector given by a point as its second argument.

```
move :: Shape -> Point -> Shape
```

It is probably a good idea to define and use an auxiliary function `addPt :: Point -> Point -> Point`, which adds two points component wise.

(e) Define a function `alignLeft` that transforms one figure into another one in which all shapes have the same `minX` coordinate but are otherwise unchanged.

```
alignLeft :: Figure -> Figure
```

Note: It might be helpful to define an auxiliary function `moveToX :: Number -> Shape -> Shape` that changes a shape's position so that its `minX` coordinate is equal to the number given as first argument.

(f) Define a function `inside` that checks whether one shape is inside of another one, that is, whether the area covered by the first shape is also covered by the second shape.

```
inside :: Shape -> Shape -> Bool
```

Hint: Think about what one shape being inside another means for the bounding boxes of both shapes.

Note that this remark is meant to help with some cases, but it doesn't solve all.