CS 381, Spring 2024 — Homework 1 (Elm)

Important Reminders!

1. Upload your solution as an Elm file (ending in `.elm`) to Canvas.
2. **Only submit files that compile without errors!** (Put all non-working parts in comments.)
3. You must do all homework assignments by yourself, without the help of others. Also, you must not use services such as Chegg or Course Hero. If you need help, simply ask on Canvas, and we will help!
4. The homework is graded leniently, and we reward serious efforts, even when you can't get a correct solution.
5. You can work in teams of up to four students to create and submit homework assignments. Groups must be set up in advance and will have to be the same over the whole term. Each group submits one solution, and all group members receive the same grade for the homework.

Download the file `HW1_Def.elm`, and insert the following lines at the top of your file (which should have the name `HW1_Elm.elm`).

```
module HW1_Elm exposing  (..)

import HW1_Def exposing  (..)
```

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 alias Bag a = List (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 : a -> Bag a -> Bag a
```

(b) Define the function `del` that removes a single element from a multiset. Note that deleting 3 from $\{2, 3, 3, 4\}$ yields $\{2, 3, 4\}$ whereas deleting 3 from $\{2, 3, 4\}$ yields $\{2, 4\}$.

```
del : a -> Bag a -> Bag a
```

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

```
bag : List 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 : Bag a -> Bag a -> Bool
```

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

(e) Define a function `isSet` that tests whether a bag is actually a set, which is the case when each element occurs only once.

```
isSet : Bag a -> Bool
```

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

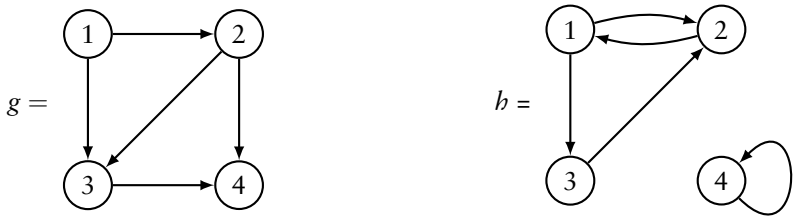
```
size : Bag a -> Int
```

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 alias Node  = Int
type alias Edge  = (Node,Node)
type alias Graph = List Edge
```

(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.



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 `asSet` (defined in the file `HW1_Def.elm`) to remove duplicates from a list and sort it.

(a) Define the function `nodes : Graph -> List 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 -> List 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)]`.

Note: All functions can be succinctly implemented by using the higher-order list functions **`map`** and **`filter`**.

Exercise 3. Programming with Data Types

The following definitions introduce a data type for representing a few basic shapes. A figure is a collection of shapes (represented as a list). 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 alias Number = Int
type alias Point  = (Number,Number)
type alias Length = Number

type Shape = Pt Point
           | Circle Point Length
           | Rect Point Length Length

type alias Figure = List Shape
type alias 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 first argument.

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

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

A Note on Testing Your Function Definitions

To test your function definitions, you have to import the module `HW1_Elm.elm` into the Elm REPL, like so.

```
> import HW1_Elm exposing (..)
```

At this point the definitions from `HW1_Def.elm` have been imported only into your file `HW1_Elm.elm`, but they are not imported into the REPL. Thus, to define test values such as `f` in the last exercise, you have to do one of two things.

(1) Define test values within `HW1_Elm.elm`. For example, add the following definition at the bottom of your file.

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

These test values are then available in the REPL when you import `HW1_Elm.elm`.

(2) Import `HW1_Def.elm` into the REPL, that is, after importing `HW1_Elm.elm`, write:

```
> import HW1_Def exposing (..)
```

Then all the definitions from `HW1_Def.elm` are available directly in the REPL as well, and you can use constructors of the `Shape` type directly. For example:

```
width (Pt (4,4))
0 : Length
```

You can then also define test values such as `f` directly in the REPL.

```
> f = [Pt (4,4), Circle (5,5) 3, Rect (3,3) 7 2]
[Pt (4,4),Circle (5,5) 3,Rect (3,3) 7 2]
    : List Shape
> map width f
[0,6,7] : List Length
```

However, these definitions are lost when you restart the REPL.