HOMEWORK 1 (HASKELL)

CS 381, Spring 2019

APRIL 16, 2019

PREPARED BY

ADAM STEWART *(stewaada)*
ANISH ASRANI *(asrania)*
LUCAS FREY *(freyl)*
MAZEN ALOTAIBI *(alotaima)*

CONTENTS

1 Exercise 1. Programming with Lists

1.1 Define the function ins that inserts an element into a multiset

```
ins :: Eq a => a -> Bag a -> Bag a
ins new_elem [] = [(new_elem, 1)]
ins new_elem ((value, counter):tail) =
    if new_elem == value then do [(value, counter+1)] ++ tail
    else do [(value, counter)] ++ ins new_elem tail
```

1.2 Define the function del that removes an element from a multiset

```
del :: Eq a => a -> Bag a -> Bag a
del target [] = []
del target ((value, counter):tail) =
    if value == target then do
        if counter <= 1 then do tail
        else do [(value, counter-1)] ++ tail
    else do [(value, counter)] ++ del target tail
```

1.3 Define a function bag that takes a list of values and produces a multiset representation

```
bag :: Eq a => [a] -> Bag a
bag [] = []
bag (head:tail) = ins head (bag tail)
```

1.4 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
subbag [] [] = True
subbag ((value, counter):tail) [] = False
subbag [] set = True
subbag ((value, counter):tail) set =
    if del value set == set then do False
    else do
        if counter == 1 then do subbag tail set
        else do subbag ([(value, counter-1)] ++ tail) (del value set)
```

1.5 Define a function isbag that computes the intersection of two multisets

```
isbag :: Eq a => Bag a -> Bag a -> Bag a
isbag [] [] = []
isbag [] set_2 = []
isbag set_1 [] = []
isbag ((value, counter):tail) set_2 =
    if subbag ([(value, counter)] ++ tail) set_2 then do ([(value, counter)] ++ tail)
    else if subbag [(value, counter)] set_2 then do isbag (tail ++ [(value, counter)]) ((del value set_2
    ) ++ [(value, counter)])
    else do
        if counter == 1 then do isbag tail set_2
        else do isbag ([(value, counter-1)] ++ tail) set_2
```

1.6 Define a function size that computes the number of elements contained in a bag

```
size :: Bag a -> Int
size [] = 0
size set = sum (map snd set)
```

2 Exercise 2. Graphs

2.1 Define the function nodes :: Graph -> [Node] that computes the list of nodes contained in a given graph

```
nodes :: Graph -> [Node]
nodes [] = []
nodes ((source,destination):tail) = norm([source,destination] ++ nodes tail)
```

2.2 Define the function suc :: Node -> Graph -> [Node] that computes the list of successors for a node in a given graph

```
suc :: Node -> Graph -> [Node]
suc node [] = []
suc node ((source,destination):tail) =
    if node == source then do norm([destination] ++ suc node tail)
    else do suc node tail
```

2.3 Define the function detach :: Node -> Graph -> Graph that removes a node together with all of its incident edges from a graph

```
detach :: Node -> Graph -> Graph
detach node [] = []
detach node ((source,destination):tail) =
    if node == source || node == destination then do detach node tail
    else do [(source,destination)] ++ detach node tail
```

2.4 Define the function cyc :: Int -> Graph that creates a cycle of any given number

```
cyc :: Int -> Graph
cyc 0 = []
cyc 1 = [(1,1)]
cyc num = zip ([1..num]) (tail ([1..num] ++ [1]))
```

3 Exercise 3. Programming with Data Types

3.1 Define the function width that computes the width of a shape

```
width :: Shape -> Length
width (Pt _) = 0
width (Circle _ r) = r*2
width (Rect _ wdth _) = wdth
```

3.2 Define the function bbox that computes the bounding box of a shape

```
bbox :: Shape -> BBox
bbox (Pt point) = (point,point)
bbox (Circle (x,y) r) = ((x-r, y-r), (x+r, y+r))
bbox (Rect (x,y) wdth hght) = ((x,y), (x+wdth, y+hght))
```

3.3 Define the function minX that computes the minimum x coordinate of a shape

```
minX :: Shape -> Number
minX (Pt (x,_)) = x
minX (Circle (x,_) r) = x - r
minX (Rect (x,_) _ _) = x
```

3.4 Define a function move that moves the position of a shape by a vector given by a point as its second argument

```
addPt :: Point -> Point -> Point
addPt (x_1,y_1) (x_2,y_2) = (x_1 + x_2, y_1 + y_2)

move :: Shape -> Point -> Shape
move (Pt point_1) point_2 = (Pt (addPt point_1 point_2))
move (Circle point_1 r) point_2 = (Circle (addPt point_1 point_2) r)
move (Rect point_1 wdth hght) point_2 = (Rect (addPt point_1 point_2) wdth hght)
```

3.5 Define a function alignLeft that transforms one figure into another one in which all shapes have the same minX coordinate but are otherwise unchanged

```
moveToX :: Number -> Shape -> Shape
moveToX new_x (Pt (_,y)) = Pt (new_x,y)
moveToX new_x (Circle (_,y) r) = Circle (new_x,y) r
moveToX new_x (Rect (_,y) wdth hght) = Rect (new_x,y) wdth hght

alignLeft :: Figure -> Figure
alignLeft [] = []
alignLeft figure = map (moveToX (minimum(map minX figure))) figure
```

3.6 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

```
sqr :: Number->Number
sqr num = num*num

inside :: Shape -> Shape -> Bool
inside (Pt (x_1,y_1)) (Pt (x_2,y_2)) =
    if x_1 == x_2 && y_1 == y_2 then do True
    else do False

inside (Pt (x_1, y_1)) (Circle (x_2,y_2) r) =
    if ceiling(sqrt (fromIntegral ( sqr (x_1 - x_2) + sqr (y_1 - y_2)))) <= r then do True
    else do False

inside (Pt (x_1, y_1)) (Rect (x_2,y_2) wdth hght) =
    if x_1 >= x_2 && x_1 <= (x_2 + wdth) && y_1 >= y_2 && y_1 <= (y_2 + hght)  then do True
    else do False

inside (Circle (x_1,y_1) r) (Pt (x_2, y_2)) =
    if x_1 == x_2 && y_1 == y_2 && r == 0 then do True
    else do False

inside (Circle (x_1,y_1) r_1) (Circle (x_2,y_2) r_2) =
```

```
    if ceiling(sqrt (fromIntegral( sqr (x_2 - x_1) + sqr (y_2 - y_1)))) > (r_1 + r_2) then do False
    else if ceiling(sqrt (fromIntegral( sqr (x_2 - x_1) + sqr (y_2 - y_1)))) <= (abs (r_1 - r_2)) then
    do True
    else do False

inside (Circle (x_1,y_1) r) (Rect (x_2,y_2) wdth hght) =
    if (x_1 - r) >= x_2 && (y_1 - r) >= y_2 && (x_2 + r) <= (x_2 + wdth) && (y_2 + r) <= (y_2 + hght)
    then do True
    else do False

inside (Rect (x_1,y_1) wdth hght) (Pt (x_2, y_2)) =
    if x_1 == x_2 && y_1 == y_2 && wdth == 0 && hght == 0 then do True
    else do False

inside (Rect (x_1,y_1) wdth hght) (Circle (x_2,y_2) r) =
    if inside (Pt (x_1,y_1)) (Circle (x_2,y_2) r) &&
    inside (Pt (x_1+wdth,y_1)) (Circle (x_2,y_2) r) &&
    inside (Pt (x_1,y_1+hght)) (Circle (x_2,y_2) r) &&
    inside (Pt (x_1+wdth,y_1+hght)) (Circle (x_2,y_2) r) then do True
    else do False

inside (Rect (x_1,y_1) wdth_1 hght_1) (Rect (x_2, y_2) wdth_2 hght_2) =
    if inside (Pt (x_1 ,y_1)) (Rect (x_2, y_2) wdth_2 hght_2) &&
    inside (Pt (x_1+wdth_1, y_1)) (Rect (x_2, y_2) wdth_2 hght_2) &&
    inside (Pt (x_1, y_1+hght_1)) (Rect (x_2, y_2) wdth_2 hght_2) &&
    inside (Pt (x_1+wdth_1, y_1+hght_1)) (Rect (x_2, y_2) wdth_2 hght_2) then do True
    else do False
```