Introduction to Programming: Assignment 2

Due: November 15, 2021. 11.59 pm

Important Instructions:

- Submit your solution in a single file named `loginid.2.hs` on Moodle. For example, if I were to submit a solution, the file would be called `spsuresh.2.hs`.
- I have provided a `template.hs` file, where I have stated the function names and types. Please do not change them. You may define auxiliary functions in the same file, but these function names should not be changed, and you must provide the proper definition for each of these.
- Please remember to rename your file to `loginid.2.hs` before submitting.
- Deviation from the instructions will result in marks being reduced.
- If your Haskell file does not compile, or if your function names differ from the ones I have specified (ensure that you use the exact sequence of small letters and capital letters), you will receive no marks.

---

1. A *run* in a list a maximal segment of contiguous elements that are all equal. Define a function `shrink :: Eq a ⇒ [a] → [a]` with the following behaviour – `shrink l` replaces each run with just a single copy of the element that repeats.

   Sample cases:

   ```
   shrink []                          = []
   shrink [1,2,3]                     = [1,2,3]
   shrink [1,1,1,2,2,3,3,3,3,4,4]     = [1,2,3,4]
   shrink [1,2,2,3,3,4,3,2]           = [1,2,3,4,3,2]
   shrink "aaabbbaaacccaaddaaa"       = "abacada"
   ```

2. Define a related function `shrink' :: Eq b ⇒ (a → b) → [a] → [a]` with the following behaviour – `shrink' l` replaces each maximal segment `xi ... xj` such that `f xi = f x(i+1) = ... = f xj` with `xi`.

   One nice application of `shrink'` is to define a version of `nub`. The Haskell library function `nub :: Eq a ⇒ [a] → [a]`, defined in `Data.List`, removes duplicates from a list, retaining only the first occurrence of each element, but it takes time proportional to n^2, where n is the length of the list. An alternative, `myNub :: Ord a ⇒ [a] → [a]`, which works in time proportional to $n \log n$, can be defined as follows:

```
myNub :: Ord a ⇒ [a] → [a]
myNub = map snd . sortOn fst . shrink' snd .
                    sortOn snd . zip [0..]
```

The function `sortOn :: Ord b ⇒ (a → b) → [a] → [a]`, defined in `Data.List`, has the following behaviour: `sortOn h l` produces a reordering `l' = [y1, ..., yn]` of `l` such that `h y1 ⩽ h y2 ⩽ ... ⩽ h yn`. The above definition of `myNub` works in time proportional to $n \log n$ because `sortOn` runs in that time, and ideally, your `shrink'` (as well as `shrink` in the previous problem) runs in time proportional to the length of the list.

Sample cases for `shrink'`:

```
shrink' snd []
    = []
shrink' snd [(1,1),(2,2),(3,3)]
    = [(1,1),(2,2),(3,3)]
shrink' snd (zip [0..] [1,1,1,2,2,3,3,3,3,4,4])
    = [(0,1),(3,2),(5,3),(9,4)]
shrink' fst (zip [10,9..] [1,2,2,3,3,4,3,2])
    = [(10,1),(9,2),(8,2),(7,3),(6,3),(5,4),(4,3),(3,2)]
shrink' id "aaabbbaaacccaaddaaa"
    = "abacada"
```

3. Knights are chess pieces whose moves are commonly characterized as two and a half squares – they move two squares in one direction and one square in an orthogonal direction. On a chessboard, rows are called *ranks* and columns are called *files*. The square on the x^{th} rank and the y^{th} file is represented by the pair (x,y). The set of squares on an 8×8 chessboard is thus given by

   ```
   squares = [(x,y) | x ← [0..7], y ← [0..7]]
   ```

   Define a function `knightMove :: (Int, Int) → Int → [(Int, Int)]` with the behavior that `knightMove (x, y) n` gives the list of squares where the knight could be in after **exactly** `n` moves, starting from the square `(x,y)`. Make sure that your list has no duplicates, and is lexicographically sorted. (*Hint*: You can use the library function `sort` defined in `Data.List`. If you think about the solution right, you will end up using the `shrink` function used earlier for removing duplicates.)

   Sample cases:

   ```
   knightMove (0,0) 0  =   [(0,0)]
   ```

```
knightMove (0,0) 1  =   [(1,2),(2,1)]
knightMove (0,0) 2  =   [(0,0),(0,2),(0,4)
                        ,(1,3)
                        ,(2,0),(2,4)
                        ,(3,1),(3,3)
                        ,(4,0),(4,2)
                        ]
knightMove (0,0) 3  =   [(0,1),(0,3),(0,5)
                        ,(1,0),(1,2),(1,4),(1,6)
                        ,(2,1),(2,3),(2,5)
                        ,(3,0),(3,2),(3,4),(3,6)
                        ,(4,1),(4,3),(4,5)
                        ,(5,0),(5,2),(5,4)
                        ,(6,1),(6,3)
                        ]
knightMove (2,2) 2  =   [(0,2),(0,6)
                        ,(1,1),(1,3),(1,5)
                        ,(2,0),(2,2),(2,4),(2,6)
                        ,(3,1),(3,3),(3,5)
                        ,(4,2),(4,6)
                        ,(5,1),(5,3),(5,5)
                        ,(6,0),(6,2),(6,4)
                        ]
```

4. The setting is the same as above, but this time we want to define `knightMove'`, with the behavior that `knightMove' (x, y) n` gives the list of squares where the knight could be in after **at most** `n` moves, starting from the square `(x,y)`. Once again make sure that your list has no duplicates, and is lexicographically sorted.

   Sample cases:

```
knightMove' (0,0) 0 =   [(0,0)]
knightMove' (0,0) 1 =   [(0,0),(1,2),(2,1)]
knightMove' (0,0) 2 =   [(0,0),(0,2),(0,4)
                        ,(1,2),(1,3)
                        ,(2,0),(2,1),(2,4)
                        ,(3,1),(3,3)
                        ,(4,0),(4,2)
                        ]
```

```
knightMove' (0,0) 3 =   [(0,0),(0,1),(0,2),(0,3),(0,4),(0,5)
                        ,(1,0),(1,2),(1,3),(1,4),(1,6)
                        ,(2,0),(2,1),(2,3),(2,4),(2,5)
                        ,(3,0),(3,1),(3,2),(3,3),(3,4),(3,6)
                        ,(4,0),(4,1),(4,2),(4,3),(4,5)
                        ,(5,0),(5,2),(5,4)
                        ,(6,1),(6,3)
                        ]
knightMove' (2,2) 2 =   [(0,1),(0,2),(0,3),(0,6)
                        ,(1,0),(1,1),(1,3),(1,4),(1,5)
                        ,(2,0),(2,2),(2,4),(2,6)
                        ,(3,0),(3,1),(3,3),(3,4),(3,5)
                        ,(4,1),(4,2),(4,3),(4,6)
                        ,(5,1),(5,3),(5,5)
                        ,(6,0),(6,2),(6,4)
                        ]
```

5. Define a function `tuples :: [[a]] → [[a]]` which, given a list of lists `[l1, ... , ln]` as input, produces the list of all tuples `[x1, ... , xn | each xi ← li]`.

   Sample cases:

```
tuples [[1,2],[3,4]]
    = [[1,3],[1,4],[2,3],[2,4]]
tuples [[1,2,3],[3,4,5]]
    = [[1,3],[1,4],[1,5],[2,3],[2,4],[2,5],[3,3],[3,4],[3,5]]
tuples [[1,2],[3,4],[2,4]]
    = [[1,3,2],[1,3,4],[1,4,2],[1,4,4]
      ,[2,3,2],[2,3,4],[2,4,2],[2,4,4]
      ]
```

6. Define a function `injTuples :: Eq a ⇒ [[a]] → [[a]]` such that

```
injTuples [l1, ... ,ln] = [x1, ... , xn | each xi ← li,
                                          i /= j ⇒ xi /= xj]
```

   Sample cases: Sample cases:

```
injTuples [[1,2],[3,4]]
    = [[1,3],[1,4],[2,3],[2,4]]
```

```
injTuples [[1,2,3],[3,4,5]]
    = [[1,3],[1,4],[1,5],[2,3],[2,4],[2,5],[3,4],[3,5]]
injTuples [[1,2],[3,4],[2,4]]
    = [[1,3,2],[1,3,4],[1,4,2],[2,3,4]]
```

7. One nice use of `injTuples` is in computing basketball lineups. In basketball, there are five positions – *point guard*, *shooting guard*, *small forward*, *power forward*, and *center*. Each team needs to have five players, and traditional lineups have one player in each position. Some players are very versatile and can play in multiple positions. When we compute lineups, we need to apply `injTuples` to ensure that we actually pick five players in each lineup. You can see the result of the function on the following data.

```
pg =    ["LeBron", "Russ", "Rondo", "Nunn"]
sg =    ["Monk", "Baze", "Bradley", "Reaves",
         "THT", "Ellington", "LeBron"]
sf =    ["LeBron", "Melo", "Baze", "Ariza"]
pf =    ["LeBron", "AD", "Melo"]
c  =    ["AD", "DJ", "Dwight"]

traditionalLineUps :: [[String]]
traditionalLineUps = injTuples [pg, sg, sf, pf, c]
```

   You will see that `length traditionalLineUps` is `547`, and it is a nightmare for the coach to pick out meaningful lineups from so many possibilities. One could reduce the number of possibilities by a less traditional lineup, consisting of three *front court players* and two *back court players*. Some players like LeBron can fit in both the front court and back court, but cannot be included twice in the same lineup. So your task is define the following function:

```
lineUps :: Eq a ⇒ [(Int, [a])] → [[a]]
lineUps [(m1,l1), ... ,(mk,lk)] returns a list of lineups.
Each lineup is of the form l1' ++ ... + lk',
    each li' consists of mi distinct players from li,
    for i /= j, li' is disjoint from lj'
```

   Here is a sample case:

```
fc1, bc1 :: [String]
fc1 = ["LeBron", "AD", "Dwight", "Melo", "THT"]
bc1 = ["LeBron", "Russ", "Baze"]
```

```
lineUps [(3,fc1), (2,bc1)] =
    [["LeBron","AD","Dwight","Russ","Baze"]
    ,["LeBron","AD","Melo","Russ","Baze"]
    ,["LeBron","AD","THT","Russ","Baze"]
    ,["LeBron","Dwight","Melo","Russ","Baze"]
    ,["LeBron","Dwight","THT","Russ","Baze"]
    ,["LeBron","Melo","THT","Russ","Baze"]
    ,["AD","Dwight","Melo","LeBron","Russ"]
    ,["AD","Dwight","Melo","LeBron","Baze"]
    ,["AD","Dwight","Melo","Russ","Baze"]
    ,["AD","Dwight","THT","LeBron","Russ"]
    ,["AD","Dwight","THT","LeBron","Baze"]
    ,["AD","Dwight","THT","Russ","Baze"]
    ,["AD","Melo","THT","LeBron","Russ"]
    ,["AD","Melo","THT","LeBron","Baze"]
    ,["AD","Melo","THT","Russ","Baze"]
    ,["Dwight","Melo","THT","LeBron","Russ"]
    ,["Dwight","Melo","THT","LeBron","Baze"]
    ,["Dwight","Melo","THT","Russ","Baze"]
    ]

fc2, bc2 :: [String]
fc2 = ["LeBron", "Wade", "Bosh", "Haslem"]
bc2 = ["LeBron", "Wade", "Allen"]

lineUps [(3,fc2), (2,bc2)] =
    [["LeBron","Bosh","Haslem","Wade","Allen"]
    ,["Wade","Bosh","Haslem","LeBron","Allen"]
    ]
```