Assignment 9 (14.01.2022)

Handin until: Friday, 21.01.2022, 00:00

Exercise 1: Functor Instances **[4 Points]**

1. Make the following data type for rose trees an instance of `Functor`:

```
data RoseTree a = RoseTree a [RoseTree a]
```

2. Consider a data type for simple key-value maps:

```
data Map k v = Map [(k, v)]
```

The following instance cannot be defined:

```
> instance Functor Map where
error:
    * Expecting one more argument to 'Map'
      Expected kind '* -> *', but 'Map' has kind '* -> * -> *'
    * In the first argument of 'Functor', namely 'Map'
      In the instance declaration for 'Functor Map'
```

Explain the problem. Define a slightly different but reasonable `Functor` instance for `Map`, instead.

Exercise 2: Merging Maps **[16 Points]**

This task is based on a problem that was part of the ACM Programming Contest World Finals in 2004. You can find the original task description here:

`http://db.inf.uni-tuebingen.de/staticfiles/teaching/ws0809/fun-problems/Merging-Maps.pdf`

Read the first page of the task description precisely before you go on with this exercise!

Merging Maps Example The following example of five maps (each with 5 columns and 3 rows) is taken from the task description:

```
      --A-C   C----   C----   ----D   -D--C
      ----D   D---F   -----   -E--B   ----G
      ----B   B----   B-A-C   -----   ----B

Map #   ❶       ❷       ❸       ❹       ❺
```

Merging them into a single map works as follows:

1. A merge of map ❶ and map ❷ with *offset* (0,4) – same row, shifted four columns right – has a *score* of 3 and results in a new map ❻ which is shown below.
2. In a second step, map ❸ is merged with map ❺ (*offset* (0,−4), *score* 2) to map ❼.
3. Next, map ❻ is merged with map ❹ (*offset* (1,0), *score* 2) to map ❽.
4. And finally, map ❼ and map ❽ are merged (*offset* (2,4), *score* 2) to the result map ❾.

```
      --A-C----   -D--C----   --A-C----   -D--C--------
      ----D---F   ----G----   ----D---F   ----G--------
      ----B----   ----B-A-C   -E--B----   ----B-A-C----
                              ---------   --------D---F
                                          -----E--B----
                                          -------------
Map #     ❻           ❼           ❽             ❾
```

Implementation in Haskell The file `MergingMaps.hs` provides some data types to be used in your implementation, together with some helper functions to read and display maps. Maps are represented with the following product type:

```haskell
data Map = Map Int Int (M.Map (Int,Int) Char)
  deriving (Show, Eq)
```

A map constructed with (`Map` *r c fs*) has a height of *r* rows, a width of *c* columns and contains the major features *fs*. Major features are stored in a dictionary (`M.Map (Int,Int) Char`, import from `Data.Map`) from coordinates (*row*,*col*) to feature values (characters `[A..Z]`).
Function `fromCharMap` that converts a string representation of a map to `Map`, and the five example maps from above are provided in `MergingMaps.hs`. Implement the merging of maps as follows:

1. Write a function `possibleOffsets :: Map -> Map -> [Offset]` that computes the **offsets** of all possible (non-disjoint) overlays of two maps.

   An (`Offset` *r c*) is a map shift of *r* rows right (left if negative) and *c* columns down (up if negative).

   For instance, for maps ❶ and ❷ we get the following list of possible offsets (note that the `Show` instance for `Offset` renders offsets like tuples).

   ```
   *MergingMaps> possibleOffsets (fromCharMap m1) (fromCharMap m2)
   [ (-2,-4),(-2,-3),(-2,-2),(-2,-1),(-2,0),(-2,1),(-2,2),(-2,3),(-2,4)
   , (-1,-4),(-1,-3),(-1,-2),(-1,-1),(-1,0),(-1,1),(-1,2),(-1,3),(-1,4)
   , (0,-4),(0,-3),(0,-2),(0,-1),(0,0),(0,1),(0,2),(0,3),(0,4),(1,-4)
   , (1,-3),(1,-2),(1,-1),(1,0),(1,1),(1,2),(1,3),(1,4),(2,-4),(2,-3)
   , (2,-2),(2,-1),(2,0),(2,1),(2,2),(2,3),(2,4)]
   ```

   Hint: Use list comprehensions!

2. Write a function `computeScore :: Map -> Map -> Offset -> Maybe Score` to compute the **score** of an overlay of two maps.

   Given two maps m_1 and m_2, and an offset `o`, `computeScore` will count the number of equal major features that overlap for an overlay of m_1 and m_2 shifted by `o`.

   Remember that two maps must not be merged if different major features overlap (collision with `'-'` is fine), or if the score would be 0. In both cases `computeScore` will return `Nothing`.

   Example: For maps ❶ and ❷ with an offset of (0,4) the score is `Just 3` because the three major features `C, D, B` overlap. With an offset (1,4), `Nothing` would be returned because a collision of `B` and `C`.

3. Given two maps, we are only interested in the best possibilities for merging, *i.e.* the offsets with the highest score:

   `bestOffsets :: Map -> Map -> Maybe (Score, [Offset])`

   If there are no possible merges for the two maps, `bestOffsets` returns `Nothing`.

   Example: For maps ❶ and ❸, there are two offsets with the best score of 2.

4. To choose exactly one merge from all `bestOffsets` the following rules apply:

   (a) Choose the offset with the minimal row offset.

   (b) If all offsets have the same row offset, choose the one with the minimal column offset.

   Define a function that chooses one of the offsets returned by `bestOffsets`:

   `bestOffset :: Map -> Map -> Maybe (Score, Offset)`

   Hint: `Maybe` is a `Functor`.

5. Now, implement the function `merge :: Map -> Map -> Offset -> Map` that performs the merge of two maps with a given offset, just as described in the introductory example.

   Hints:

   (a) You might want to use `M.mapKeys` and `M.union`.

   (b) Make sure that your function deals with negative offsets correctly!

6. To tie things up, implement the main function `mergeMaps :: [Map] -> [Map]` that iteratively merges maps into larger and larger maps (two maps at a time). In each round, choose the best two map candidates for merging by selecting the pair of maps with the best score and merge them with the corresponding offset.

   Merging stops once there is either only one map left or no more maps can be merged (for example because all scores are 0). All remaining maps are returned.

```
*MergingMaps> putStrLn $ showMaps $ mergeMaps $ map fromCharMap maps
-D--C--------
----G--------
----B-A-C----
--------D---F
-----E--B----
-------------
```