Programming in Haskell – Midterm Exam

UNIZG FER, 2015/2016

Note: Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message.

Each problem is worth a certain number of points. The points are given at the beginning of each problem, and are scaled to 10 upon grading.

You need at least *5 points* (after scaling) and at least *one correctly solved task* to pass this midterm exam. You have one hour for solving this exam. You are to solve it on your laptop and may use the Internet, save for communicating with one another, or third parties.

1. *(2 pts)* After the wild success of the `finalRanking` function in the last Nintendo World Championship, a need has arisen for a more general application of the same functionality. This more general function should work for any number of contestants, instead of the three used in the first version.

   Define the `ranking` function that takes the names and scores of any number of contestants as 2-tuples `(name, score)` and returns them ranked, in the form of a list of strings. The players are ranked first by score (a higher score is better) and, if their scores are equal, by name, using lexicographical ordering.

   The list of names should be returned so that the best-ranked player is first and the worst-ranked player is last in the return value.

   You may use any language construct or function you deem necessary.

   ```
   ranking :: [(String, Int)] -> [String]
   ranking [("John", 20), ("Paul, 360), ("Ringo", 10)]
   ⇒ ["Paul", "John", "Ringo"]
   ranking [("Eden", 1000), ("Christiano", 500), ("Xavier", 460), ("Leo",
   500)] ⇒ ["Eden", "Christiano", "Leo", "Xavier"]
   ranking [("Tom", 1] ⇒ ["Tom"])
   ranking [] ⇒ []
   ```

2. *(2 pts)* Define the function `zipN` that behaves like `zip`, but operates on an arbitrary number of arguments of the same type, arranged in a single input list.

   ```
   zipN :: [[a]] -> [[a]]
   zipN ["abcdef", "12345678"] ⇒ ["a1", "b2", "c3", "d4", "e5", "f6"]
   zipN ["abc", "banana", "crocodile"] ⇒ ["abc", "bar", "cno"]
   zipN [[1..3], [2..], [3..]] ⇒ [[1,2,3], [2,3,4], [3,4,5]]
   ```

```
zipN [] ⇒ []
```

3. *(2 pts)* Define a function `mapWhile` that takes a predicate, a unary function and a list of elements. It works similarly to `map`, but instead of applying the function once, it applies it for as long as the predicate is true for that element of the list.

```
mapWhile :: (a -> Bool) -> (a -> a) -> [a] -> [a]
mapWhile (<3) (+1) [1..5] ⇒ [3,3,3,4,5]
mapWhile isSpace (const '-') "Some text here" ⇒ "Some-text-here"
mapWhile (any (<0)) (map (+100)) [[-1,0,1], [99,100]]
⇒ [[99,100,101], [99,100]]
mapWhile (const True) (id) [] ⇒ []
mapWhile (cost True) (id) [1] ⇒ [⊥]
```