CS-205 Declarative Programming
Cécilia Pradic, Monika Seisenberger

October 30th 2023

CS-205 coursework: due November 13th 2023, 11am

1. Submission for this coursework will be individual; detailed submission instructions will be given on Canvas next week.
2. Automated tests will be used to award marks; you should scrupulously follow instructions. In particular, the types of your functions and their names should match exactly what is in this document (case included).
3. The examples input-output behaviour in the handout are just there for your benefit and encourage you to systematically test your code on examples on your own; they are *not* related to the tests used for marking (they will be randomly generated in **large** numbers).
4. Feel free to indicate any further relevant information in a comment at the beginning of the file (please keep it as simple as possible).
5. You are allowed to use any function that is shipped with `ghc`. If you feel some function is so basic it should come with a standard Haskell distribution, recall that you can look up functions by types or names on `https://hoogle.haskell.org`. You are free to add import statements at the top of your file to access those functions (like `import Data.Char` to access `isDigit`).
6. By submitting coursework electronically, you state that you fully understand and are complying with the University's policy on Academic Integrity and Academic Misconduct. The policy can be found at `https://myuni.swansea.ac.uk/academic-life/academic-misconduct`. The consequences of committing academic misconduct can be extremely serious and may have a profound effect on your results and/or further progression. The penalties range from a written reprimand to cancellation of all of your marks and withdrawal from the University.
7. There is a total of 40 marks to be gained by answering the questions. 5 additional marks will be given for adherence to the submission rules (program compiling and containing all relevant definitions; that will be checked automatically) and 5 marks for good style (e.g. proper indentation, readable code, no line longer than 100 characters and no TAB characters), so there is a total of 50 marks to be earned.

Question 1. A pizzeria sells pizzas of an arbitrary size and with an arbitrary number of toppings. The owner Giovanni wishes to have a program that allows to compute the selling price of a pizza depending on its size (given by its diameter in cm) and the number of toppings, as well as the number of sauces added. The pizza base costs £0.002 per cm^2 and the costs for each topping are £0.001 per cm^2. Each sauce costs £0.50. Since Giovanni also wants to make some profit, he multiplies the costs of a pizza by a factor 1.5. Can you help Giovanni with a suitable well-structured Haskell function

```haskell
pizzaPricing :: Float -> Int -> Int -> Float
```

where the first input is the pizza diameter, the second one is the number of toppings and the last one the number of sauces? The computed result should be truncated after two digits, e.g. if the computed price is 6.5999, then the resulting price should be 6.59.

[8 marks]

Question 2. Consider the following datatype for non-empty lists

```haskell
data NEList a = Single a | Cons a (NEList a)
  deriving Show
```

Write functions

```
fromNEList :: NEList a -> [a]
toNEList :: [a] -> Maybe (NEList a)
```

that convert between members of `NEList a` and regular haskell lists. In particular, we should have the following equalities being `True`:

```
fromNEList (Cons 2 (Single 1)) == [2, 1]
toNEList [] == Nothing
toNEList [7,8,9] == (Just . Cons 7 . Cons 8 . Single $ 9)
```

[6 marks]

Question 3. (Phonetic Search) The following is a programming test for job applicants provided by a phone company. You will write a function

```
matchNames :: [String] -> [String] -> [(String, [String])]
```

which takes two lists of surnames, say, `providedSurnames` and `allSurnames`, and output a list whose elements are all the pairs where the first component is a surname `n` from `providedSurnames` and the second component are maximal lists of of *matching surnames* from `allSurnames`.

Surnames are said to match according to the following rules:

1. All non-alphabetic characters are ignored
2. Word case is not significant
3. After the first letter, any of the following letters are discarded: A, E, I, H, O, U, W, Y.
4. The following sets of letters are considered equivalent
   - A, E, I, O, U
   - C, G, J, K, Q, S, X, Y, Z
   - B, F, P, V, W
   - D, T
   - M, N
   - All others have no equivalent
5. Any consecutive occurrences of equivalent letters (after discarding letters in step 3) are considered as a single occurrence

The rules should be applied in that order. So, if we consider the following two candidate inputs:

```
providedSurnames :: [String]
providedSurnames = ["Jones", "Smith"]

allSurnames :: [String]
allSurnames = [
                    "Smith",
                    "Smyth",
                    "Smythe",
                    "Smid",
                    "Schmidt",
                    "Smithers",
                    "Jonas",
                    "Johns",
```

```
                    "Johnson",
                    "Macdonald",
                    "Nest O'Malett",
                    "Ericsson",
                    "Erikson",
                    "Saunas",
                    "Van Damme"
                ]
```

calling `matchNames providedSurnames allSurnames` should return

```
[("Jones",["Jonas","Johns","Saunas"]),("Smith",["Smith","Smyth","Smythe","Smid","Schmidt"])]
```

[13 marks]

Question 4. (Navigating a maze)

We are considering the following scenario: two birds, blue and red, are dropped in a closed-off maze. There is one set of stairs in the middle that allow to get out. Now the birds are very smart and know the following: if you are lost in a maze, you can always find an exit if at every intersection you go to your left and go backwards only if you are forced to. Under mild assumptions, this always works and does not require knowing the whole layout of the maze.

We will assume that the blue bird is dropped at the bottom-left of the maze and is initially facing north, while the red bird is dropped at the top-right and facing south. We are now interested in which bird manages to reach the stairs first, assuming they move at exactly the same speed, use the algorithm outlined above and that they can go past each other in a corridor without any issues. In Figure 1, we picture such a situation where the blue bird reaches the stairs first.

For this question, we encode mazes as a list of walls, its size and the position of the stairs. We will thus be using the datatype `Maze` defined as follows:

```
data Orientation = H | V
  deriving (Show,Eq)
type Wall = (Int, Int, Orientation)
  data Maze = ((Int,Int),(Int, Int), [Wall])
```

A wall is given by its position of its bottom left corner on the 2D grid and an orientation (vertical or horizontal). An object of type `Maze` is a tuple `((width,height), (stairX, stairY),ws)` where the first two coordinates are the dimensions of the maze, which is assumed to start at coordinate `(0,0)` and `ws` is the list of walls of the maze. A point `(x,y)` is considered to be out of the maze if either `x >= width`, `x < 0` , `y >= height` or `y < 0`).

Example: The maze pictured in Figure 1 can be encoded in Haskell as follows:

```
exampleMaze :: Maze
exampleMaze = ((4,4), (2,2), hWalls ++ vWalls)
   where vWalls = map (\ (i,j) -> (i,j,V))
                  [
                    (0,0),(0,1),(0,2),(0,3),
                          (1,1),(1,2),
                          (2,1),(2,2),
                                (3,2),(3,3),
                    (4,0),(4,1),(4,2),(4,3)
                  ]
         hWalls = map (\ (i,j) -> (i,j,H))
                  [
                    (0,0),(1,0),(2,0),(3,0),
                    (0,1),      (2,1),
                                (2,2),
                    (0,4),(1,4),(2,4),(3,4)
                  ]
```

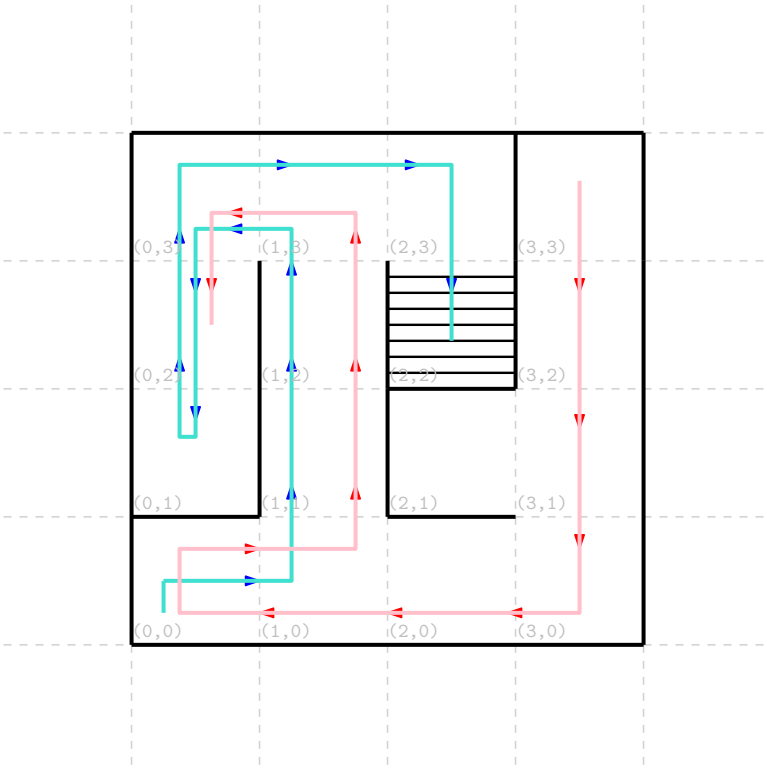

Figure 1: Picture of a maze and of two paths through it, blue and pink, obtained using the algorithm described informally above.

Task: Write a function

```
redBirdGetsOutBeforeBlueBird :: Maze -> Bool
```

that outputs `True` if the red bird gets out before the blue bird, or at the same time.

On our example maze, the output should be `False`.

To simplify matters, you can assume that the input maze is not degenerate in the sense that:

- All expected outer walls are there.
- For any point of the grid within the boundary of the maze, there is a wall touching it.

This exercise is somewhat open-ended; you should not hesitate to introduce your own helper functions and datatypes, as well as comments explaining your code. Your solution should work on any valid maze as described above.

[13 marks]