CS-205 Declarative Programming
Cécilia Pradic, Monika Seisenberger

October 30th 2024

CS-205 coursework: part 1 (functional programming)
due November 21st 2024, 11am

1. Submission for this coursework will be individual and on autograder. We aim to have the submission server by Friday 8th and more detailed submission instructions will be given on Canvas.

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. Use of "generative AI" is forbidden.

8. There is a total of 30 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 40 marks to be earned.

Question 1. (Manlio's entrepreneurship) A pizzeria sells mozarella sandwiches of an arbitrary length and with an arbitrary number of additional ingredients. The owner Manlio wishes to have a program that allows to compute the selling price of a sandwich depending on its size (given by its length in cm) and the number of additional ingredients. The baguette

costs £0.1 per cm. A mozarella slice costs £0.2 and has a diameter of 3cm; Manlio wants to put just enough mozarella slices so that no part of the sandwich is mozarella-free. The other extra ingredients are less expensive and each cost £0.2 per centimeter. Since Manlio also wants to make some profit to feed his cat, he multiplies the costs of a sandwich by a factor 1.5. As most people now want their sandwiches to be delivered, Manlio hires a delivery person that additionally takes a fee of £5 per kilometers

Can you help Manlio with a suitable well-structured Haskell function

```
sandwichPricing :: Float -> Int -> Float -> Float
```

where the first input is the length of the sandwich in centimeters, the second one is the number of extra ingredients and the last one is the distance (in kilometers) from the shop to the delivery point? 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.

Tip. *You may want to use the following functions defined in the standard library:* `fromIntegral`, `truncate` *and* `ceiling`. *Also try to make your code readable! You probably will need to use* `where` *clauses to accomplish that.*

[8 marks]

Question 2. (Lists of friends)

Tip. *For this exercise, it might be a good idea to use* list comprehensions *and investigate whether functions from the* `Data.List` *module might help.*

(a) Write a function

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

that takes as input a list of pairs indicating whether two persons are friends, the name of a person, and returns all the people who are a friend of a friend of that person. Note that (x, y) belongs to the first input if and only if x is a friend of y – in particular, this does not mean that y is a friend of x.

In particular we should have that

```
listAllFriendsOf
  [("Alice","Bob"), ("Eve", "Alice"), ("Bob", "Leah"), ("Bob", "Alice"),("Keir", "Bob")]
  "Alice"
```

should return `["Alice", "Leah"]` (up to permutation and duplication).

[4 marks]

(b) Write a function

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

that lists people who are both a friend and an enemy to someone. Like in the previous question, the first input is the list of friendships and the second lists enmities.

[4 marks]

(c) The list of friends of enemies and friends has leaked and now people are reassessing their relationships! They feel they can no longer be friends with people who are friendly to any of their enemies. Write a function

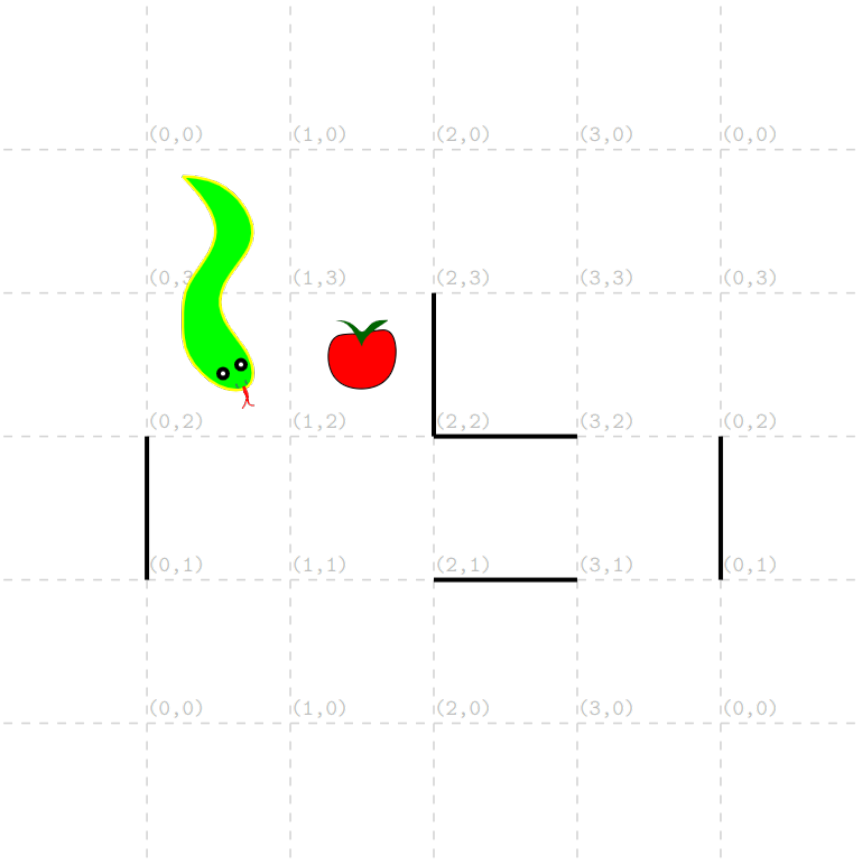

Figure 1: Initial position of a game of snake

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

taking the same inputs as before and outputting the updated list of friends.

[4 marks]

Question 3. (Snakes)

This exercise is a long one and does not give you all the intermediate steps towards solving it. You should not hesitate to introduce intermediate functions.

Let us consider the single-player variant of the popular *Snake* game. In this game, a snake, represented as a contiguous line, advances on step on a discrete grid as directed by the player. The grid will contain a fruit somewhere; when the head of the snake reached the fruit, the snake grows and another fruit spawns somewhere on the grid. The game ends when the snakes collides with its own tail, dying in the process; the goal of the game is to end with the longest possible snake (and in multiplier variants, to keep the snake alive as long as possible!).

In all of the scenarios we consider, we assume we are working with a rectangular map, with position indexed by pairs of integers (x, y). We use the following type alias to talk about positions, as well as the following type declaration to talk about the direction of the moves the snake might decide to take.

```
type Pos = (Int, Int)

data Dir = North | South | West | East
  deriving (Show, Enum, Eq)
```

We will consider that the map is *toroidal*, that is that when the snake crosses the eastern boundary at a given height, it should re-appear on the western boundary at the same height and vice-versa; similarly, crossing the northern/southern boundary results in the snake appearing on the opposite end of the grid.

Finally the map may contain some walls the snake cannot go through without dying. We use the following datatype to represent walls:

```
data Orientation = H | V
  deriving Eq

type Wall = (Pos, Orientation)
```

A wall is given by its position of its bottom left corner on the 2D grid and an orientation (vertical or horizontal).

In all cases, we will assume that the snake begins on the top left of the map, facing downwards, and is initially of length two - so, assuming h is the height of the map, the snake's head occupies the cell $(0, h-2)$ and the tail the cell $(0, h-1)$. Such a situation is pictured in Figure 1.

(a) Write a function

```
snakeFate :: Int -> Int -> [Wall] -> [Pos] -> [Dir] -> (Bool, Int)
```

that runs the game and outputs a pair consisting of

- a boolean which should be `True` if and only if the snake is still alive at the end
- an integer representing the length of the snake

when give the following arguments in order:

- the width `width` of the map
- the height `height` of the map
- the list of walls `walls` on the map (positions of walls (x, y) are stricly inferior to $(\mathtt{w}, \mathtt{h})$)
- the list `fruits` of positions where fruits should spawn
- the list of moves `moves` of the snake

Recall that at most one fruit should be spawned on the map at a time. Initially, the first element of `fruits` is available on the map. Whenever a fruit is eaten, the next fruit in the list should be spawned. However, if it is supposed to be spawned on a cell already occupied by the snake, it should not be spawned and simply discarded from the list of fruits to be spawned – one should then attempt to spawn the next fruit in the list until one is found. If the list of fruits is exhausted, no fruit should be spawned on the map.

To clarify the situation, let's look at some sample inputs and what the output should be in each case. For simplicity, we will always take `width` and `height` to be 4 and use the following constants to model walls and fruits

```
exWalls :: [Wall]
exWalls = [((0,1), V), ((2,2), V), ((2,2), H), ((2,1),H)]

exFruits :: [Pos]
exFruits = [(1,2),(1,1),(0,1),(2,2),(3,3)]
```

With these, the initial position of the game is faithfully represented by the picture given in Figure 1. We now give the examples

- ```
  let eswn = [East, South, West, North] in
    snakeFate 4 4 exWalls exFruits eswn
  ```

  should evaluate to `(False, 5)`; this corresponds to the scenario where the snake collides with itself after the last move.

- ```
  let eswsseesenn = [East, South, West, South, South, East,
                     East, South, East, North, North]
   in
    snakeFate 4 4 exWalls exFruits eswsseesenn
  ```

  should evaluate to `(True,7)`. In this scenario, the snake manages to eat all of the fruits and live.

- ```
  let senn = [South, East] ++ replicate 50 North in
    snakeFate 4 4 exWalls exFruits senn
  ```

  should evaluate to `(True,3)`. In this scenario, the snake eats the first fruit, the second fruit does not spawn because the cell is occupied by the tail of the snake (so the third fruit spawns instead on the first column) and the snake just loops on the second column for a while

- ```
  let esee = [East, South] ++ replicate 20 East in
    snakeFate 4 4 exWalls exFruits esee
  ```

  should evaluate to `(False,4)`. In this case the snake eats the first two fruits and then runs into the wall `((0,1),V)`; had the wall not been there, it would have lived!

Most marks will be awarded if your result is correct for maps which contain no walls whatsoever.

[8 marks]

(b) Now assume that there are no walls whatsoever and, aside from where the snake starts, the whole map is covered in fruits! Write a function

```
eatAllFruits :: Int -> Int -> [Dir]
```

taking as input the width and the height of the map and returning the sequence of moves the snake to eat all fruits and staying alive. You should also ensure that the snake is still able to move in the next step.

For instance, a valid output for `eatAllFruits 3 3` would be

```
[South,East,East,North,West,North,East]
```

but `[South,East,East,North,North,West,South]` would not be correct because the snake would die in the next move.

> **Tip.** *Make a case distinction according to the parity of the width and draw some pictures!*

Your solution is expected to run in time $\mathcal{O}(\texttt{width} + \texttt{height})$ – solutions that will make the grading server timeout won't earn you marks.

[2 marks]