CS-205 Declarative Programming
Cécilia Pradic, Monika Seisenberger

October 27th 2025

CS-205 coursework 1
due November 12th 2025, 11am

1. Submission for this coursework will be individual and on autograder. More detailed submission instructions will be given on Canvas. Submissions will be possible on autograder from November 3rd.

2. Automated tests will be used to award some of the 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) and the template provided on canvas.

3. The examples of 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" for completing this coursework is forbidden (and disrespectful).

8. 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. (Frøy's bakery) A bakery sells bespoke lemon cake of arbitrary dimensions. The owner Frøy wishes to have a program that allows to compute the selling price of such a cake depending on its size, the number of additional toppings, the age of the customer and the

number of blessings to be applied on the cake. One liter of cake costs £5, and the icing costs £0.3 per square centimeter. The cakes are all cube-shaped and the icing is always applied to the top face only. Frøy can then add a blessing for £2. A discount of 10% is applied for customers who are strictly over 100 years old, and then Frøy must increase the cost by 20% to account for VAT.

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

```
cakePricing :: Float -> Int -> Int -> Float
```

where the first input is the volume of the cake in liters, the second one is the number of blessings and the last one is the age of the customer.

[8 marks]

Question 2. (Author lists)

The theme of this question will be parsing and formatting lists of authors of scientific publications in the style of BibTeX[1]. The use case is that a user is going to provide as input a list of authors where first names and last names are comma-separated and authors are separated by `"and"`, much like in BibTeX, and expect outputs that could be used in bibliography lists or headers of publications. But before that, there are a few warm-up questions.

(a) Write a function

```
enumerationHumanReadable :: [String] -> String
```

that takes as input a list of strings and ouputs a string that separates them by commas, except for the second-to-last and last elements that should be separated by an `"and"`. The elements should be enumerated in the same order they are given. In particular, we should have

```
enumerationHumanReadable ["bird", "insect", "rodent"] == "bird, insect and rodent"
enumerationHumanReadable ["bird"] == "bird"
enumerationHumanReadable ["blue", "red"] = "blue and red"
```

> **Tip.** *Recall that pattern-matching can be nested like in the following declaration:*
>
> ```
> alternatingSum :: [Int] -> Int
> alternatingSum [] = 0
> alternatingSum [x] = x
> alternatingSum (x : y : xs) = x - y + alternatingSum xs
> ```

[6 marks]

(b) Write a function

```
tokenize :: String -> [String]
```

that takes as input a string that may contain spaces and returns the list of space-separated words and punctutation in the input. Consecutive spaces should be treated the same as a single space and there should be no empty string in the output. For instance we should have

[1] But simplified to make the exercise more straightforward; in particular, we are going to assume that everyone has both a first name and a last name, which is generally not true nor enforced by BibTeX.

```
tokenize "hello world!! " == ["hello", "world", "!", "!"]
tokenize "  the sun   is shiny, and yellow." ==
           ["the", "sun", "is", "shiny,", "and", "yellow", "."]
tokenize "aaa , bbb" == ["aaa", ",", "bbb"]
```

> **Tip.** *You may want to lookup the function* `words :: String -> [String]` *from the standard library for inspiration.*

[6 marks]

(c) Write a function

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

that processes a string and outputs a list of pairs `("firstName", "lastName")`. Different authors are separated by the keyword `"and"` and a single author name is given by either

- either `"lastName, firstName"`, possibly with an extra space before the comma
- or `"firstName lastName"` when `"firstName"` does not contain any spaces.

Beware that names may contain interstitial spaces (but no commas), but they have no space at the beginning or the end. You should list the names in the same order they are given. Here are some examples of inputs and outputs:

```
getAuthors "Jordon, Liam and Moser, Philippe" ==
             [("Liam", "Jordon"), ("Philippe", "Moser")]
getAuthors "Shota Murakami and Takeshi Yamazaki and Keita Yokoyama" ==
             [("Shota", "Murakami"), ("Takeshi", "Yamazaki"), ("Keita", "Yokoyama")]
getAuthors " Pauly  , Arno and  Kihara , Takayuki" ==
             [("Arno", "Pauly"), ("Takayuki", "Kihara")]
getAuthors "van   Oosten, Jaap" ==
             [("Jaap", "van Oosten")]
```

[6 marks]

Question 3. (Gray's pets)

After the description of the setting, there are a few easy subquestions to warm you up, but the last subquestion is more challenging.

Gray keeps a number of critters (also referred to as pets) in an enclosure of prescribed dimensions given by a pair of integers `(w, h)` that stands for width and height. Time and space are discrete in the enclosure, that is we only consider integer coordinates and the critters move in a turn-based fashion. x-coordinates within the enclosure range from `0` to `w-1` and y-coordinates from `0` to `h-1`. We thus define the following type synonyms for the sequel.

```
type MapSize = (Int, Int)
type Pos = (Int, Int)
```

Each critter will be located in a specific cell denoted by some element of type `Pos` but will also be looking in some specified cardinal direction as specified by the datatype `Direction` defined below.

```
data Direction = North | West | South | East
  deriving (Eq, Show, Ord, Enum, Bounded)

data Location = Location Pos Direction
  deriving (Eq, Show)
```

(a) Write a function

```
followDirection :: Direction -> Pos -> Pos
```

which takes as input a position `(x,y)` in a 2D grid, a direction and outputs a position shifted by 1 in the relevant direction, assuming that the first coordinate denotes steps eastwards and the second steps northwards. For instance,

```
followDirection West (followDirection West (3,4))
```

should evaluate to `(1,4)`.

[2 marks]

Now at each step, a critter can look at the adjacent cells in each `Direction` to see if it is currently empty and accessible. If it is not, it means that either another critter is sitting there, or it is not contained in the enclosure. With this information, the critter can decide to

- either move one cell forward
- or stay in the same cell and turn to face a new direction

To model this, we introduce a datatype to talk about orientation relative to whichever direction the critter is facing.

```
data Orientation = Forward | ToLeft | ToRight | Behind
  deriving (Eq, Show, Ord, Enum, Bounded)
```

(b) Write a function

```
changeDirection :: Orientation -> Direction -> Direction
```

such that `changeDirection o d` outputs the direction faced after turning according to `o` when starting from direction `d`. For instance, we should have `changeDirection ToLeft North == West`.

[2 marks]

We assume that the critters have no memory and are deterministic[2]. We thus assume that each pet's behaviour is determined by a function

```
ai :: (Orientation -> Bool) -> Orientation
```

whose input is a function `f` such that `f o == True` if and only if the cell on the side `o` of the critter is free. Then we have `ai f == Forward` if the pet decides to move forward, or otherwise `ai f` corresponds to the pet turning around in the prescribed orientation.

With this in mind, we define the following datatype for critters, where the first field is a name and the second field is the function that determines how the critter moves.

```
data Pet = Pet String ((Orientation -> Bool) -> Orientation)
```

Once all pets make their decisions, a number of things may happen:

- if a pet attempts to change the direction they're facing, they just do that while remaining in the same cell

[2] Due to the otherworldy powers of the enclosures.

- if a pet attempts to go forward:
  - if it attempts to go forward on a cell that is already occupied, that is *rude*. But nothing happens; it stays on the same cell, facing the same direction.
  - if it attempts to go to an unoccupied cell and no one else attempts to do so, then it does just that.
  - if multiple critters attempt to move to the same unoccupied cell, they collide, explode and are removed from the map. The explosion clears the cell instantly, and the cell is thus regarded as free for the next turn.

An extended example of such a simulation with five critters is pictured in Figure 1.

(c) Write a function

```
rude :: Pet -> Bool
```

that outputs `True` when it is possible that the input pet may attempt to move on a cell it sees to be occupied. For instance, the pet `Pet "Beaver" (\_ -> Forward)` is definitely rude, but none of the pets in Figure 1 are rude.

[2 marks]

(d) Write a function

```
whoIsUnexploded :: MapSize -> [(Pet, Location)] -> Integer -> [String]
```

which takes as input the size of the enclosure, the list of pets in the enclosure as well as their locations and an integer corresponding to a number `t` of turns we want to simulate, and outputs the list of names of the pets who are still on the map after `t` turns. You may assume that the input is well-formed in the sense that all cells starts of containing at most one pet, no pet is located outside of the enclosure and `t` is nonnegative. You may further assume that there are no rude pets in the input. For instance, assuming the definitions from Figure 1 are in the file, we should have that `whoIsUnexploded (width, height) initialBoard 1` should be equal to (a permutation of[3]) `["Anemone","Trilobite","Pterodactyl", "Cicada", "Dragonfly"]` but

```
whoIsUnexploded (width, height) initialBoard 2}
```

should be `["Pterodactyl", "Dragonfly"]`.

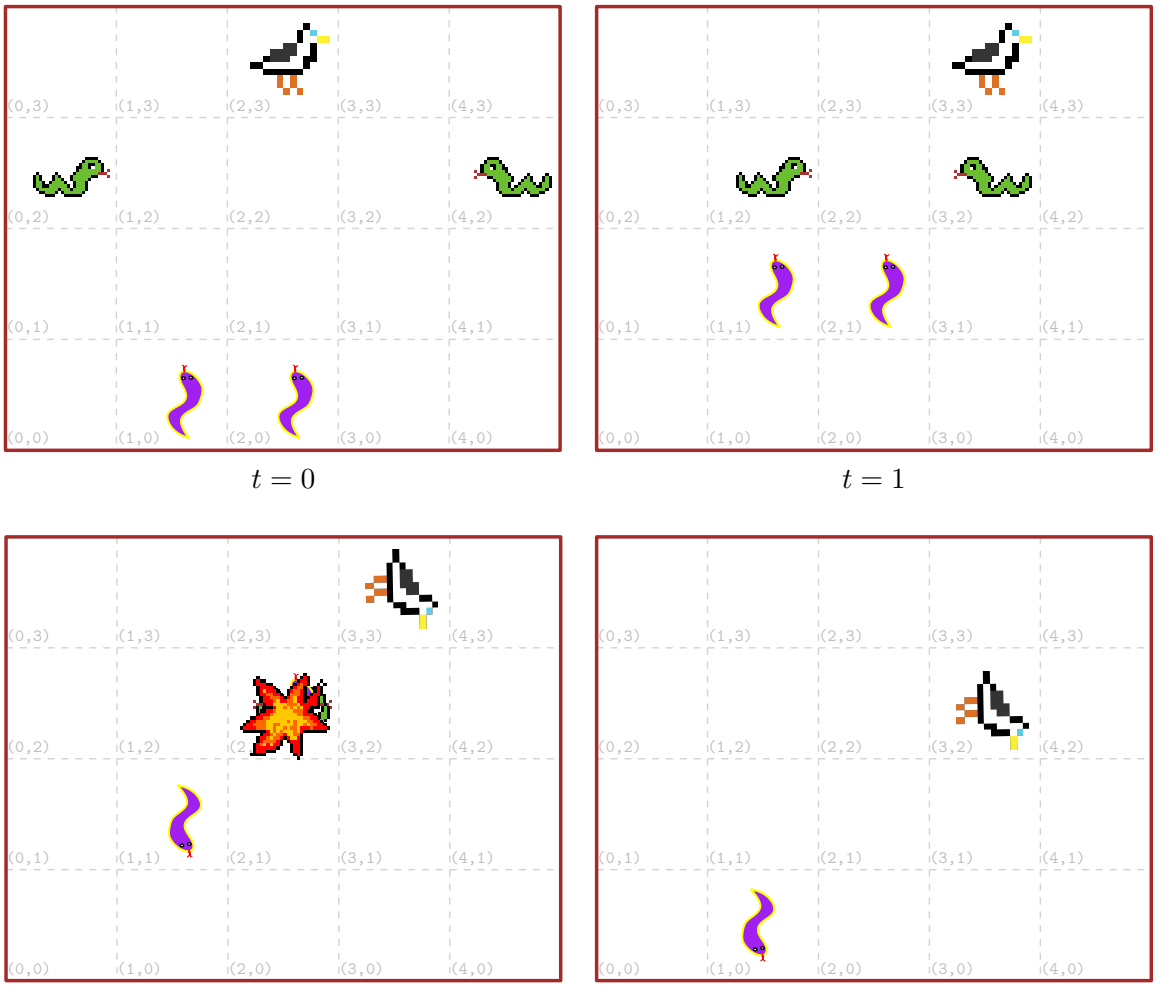
This question is more challenging than the rest. You are encouraged to introduce auxiliary definitions. It will also be mostly handgraded. Partial marks will be awarded for substantial progress towards a solution.

[8 marks]

(e) Bonus question (not marked): a naive solution of the previous subquestion usually takes time at least quadratic in the number n of pets at play. Write an algorithm that works in time $\mathcal{O}(n\log(n)(\log(w)+\log(h)))$ where w and h are the dimensions of the enclosure.

This question is substantially harder than the rest of this coursework. Do not attempt unless you are sure to have the rest 100% correct and some spare time. One idea to achieve the bound is to use quadTrees[4], and a convenient way of working with quadTrees is to use zippers[5].

[3] I don't care in which order you return the pet names, as long as they are all there and not repeated.
[4] See e.g. `https://en.wikipedia.org/wiki/Quadtree`
[5] `https://en.wikipedia.org/wiki/Zipper_(data_structure)`




```haskell
snakeAI :: (Orientation -> Bool) -> Orientation
snakeAI f | f Forward = Forward
          | otherwise = Behind

seagullAI :: (Orientation -> Bool) -> Orientation
seagullAI f | not (f ToRight) = ToRight
            | f Forward = Forward
            | otherwise = Behind

width = 5
height = 4

initialBoard = [(Pet "Anemone" snakeAI, Location (0,2) East),
                (Pet "Trilobite" snakeAI, Location (4,2) West),
                (Pet "Pterodactyl" snakeAI, Location (1,0) North),
                (Pet "Cicada" snakeAI, Location (2,0) North),
                (Pet "Dragonfly" seagullAI, Location (2,3) East)]
```

Figure 1: An example of an enclosure containing some critters. The initial state of the enclosure is descibed in the code snippet and pictured on the upper left corner. Then the successive states for the timesteps $t = 0, 1, 2, 3$ are pictured. There is an unfortunate collision at $t = 2$, which means that only Pterodactyl and Dragonfly survive.