Middle East Technical University   Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2021-2022

Programming Exam 2

Due date: 08 April 2022, Friday, 23:59

1 Problem Definition

Remember where we left off? Last time you did a great job of helping the researcher in robotics. So, your help is wanted again. Now that you know how to guide a robot, it is time to shift up to second gear. As in the previous programming exam, you will implement 6 Haskell functions in 2 parts. The same incremental manner (i.e. previous functions within the parts may be used in the implementation of rest) is applied here too.

1.1 General Specifications

- The signatures of the functions, their explanations and specifications are given in the following section. Read them carefully.
- Make sure that your implementations comply with the function signatures.
- You may define helper function(s) as you needed.
- Importing any modules is not allowed for this exam. All you need is already available in the standard `Prelude` module.
- Whenever you need to output a `Double` value (included in a list or by itself). You MUST use the following function to round it to 2 decimal places:

```
getRounded :: Double -> Double
getRounded x = read s :: Double
               where s = printf "%.2f" x
```

  This tricky implementation uses `printf` function from `Text.Printf` module. This import and the implementation itself is already included in the template file. You can use it directly, while outputting Double values. The whole purpose of using this function is the simplification of the output, which will be useful in both debugging and evaluation processes.

- Whenever needed, you can calculate a Double value as you wish. However, in case you feel overwhelmed, you can use the function below which is given in template:

```
castIntToDouble :: Integral a => a -> Double
castIntToDouble x = (fromIntegral x) :: Double
```

- A little bit differently than PE1, all of the simulations will take place in `Discretized` 2D environment with a **single** robot. Therefore, you are not expected to consider beyond. In fact, here is the type synonym we use for the points in the environment:

```
type Point = (Int, Int)
```

- Other type synonyms are given below for the dimensions of an area to explore (a specific part of the environment we are interested in) and the direction indicated by unit vectors for the robot's actions [1].

```
type Dimensions = (Int, Int)
type Vector = (Int, Int)
```

- Empty list will **not** be used in any testcases. However, you might need an empty list in the functions you implement as an edge case.
- Here is the figure for better understanding of types above and the simulation environment:

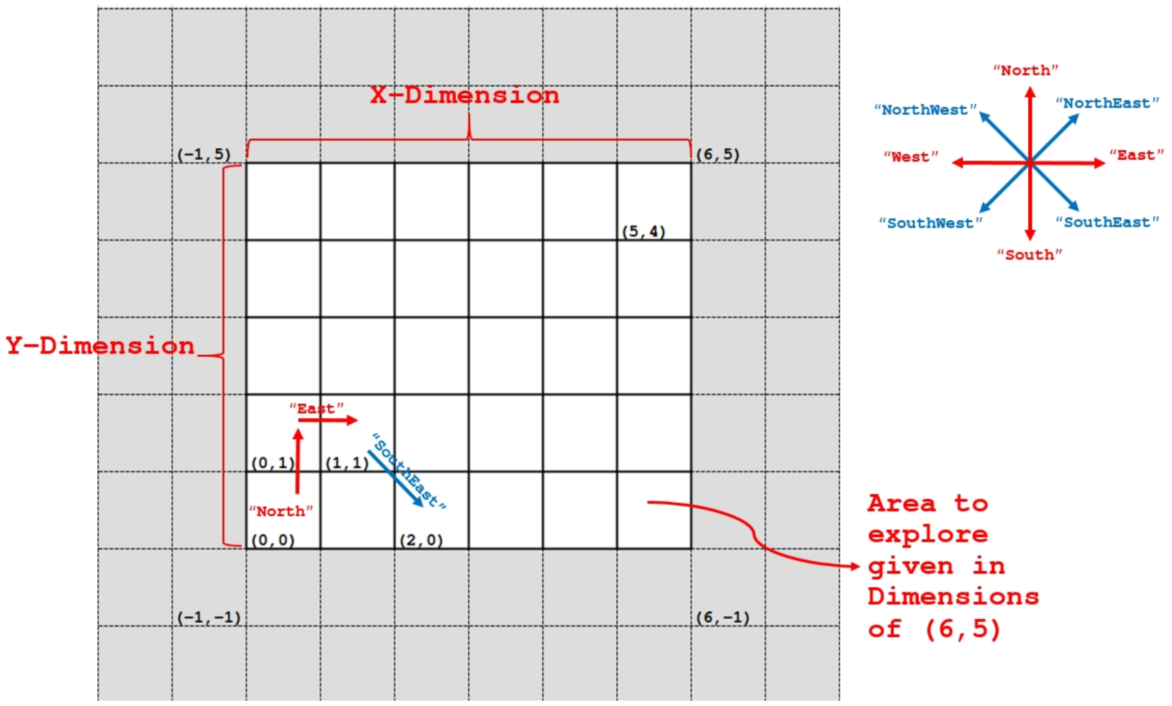

Figure 1: An example area to explore in the environment. Assuming that the robot is placed at (0,0) initially, the `North` action creates a unit vector of `(0,1)` and the robot moves to the location of `(0,1)` and then with `East` action it moves to the location of `(1,1)` and so on.

[1]You may be thinking why we keep producing names for the same type, but it really eases the understanding of a function signature.

2 Part I (45 points)

2.1 getVector (15 points)

You will implement a function named `getVector` which takes a `String` representing the robot's action and return the unit `Vector` for that action. As can be seen in the figure above, the unit `Vector` of an action is like the difference between the initial and final coordinates for a robot taking that action.

Here is the signature of this function:

```
getVector :: String -> Vector
```

Remember that there are only 9 `Strings` (i.e 8 directions and `"Stay"` from PE1) which can be given as an argument. Please make sure that you use the **correct** form of a `String` for an action in order to avoid losing points redundantly.

SAMPLE I/O:

```
*PE2> getVector "North"
(0,1)

*PE2> getVector "NorthWest"
(-1,1)

*PE2> getVector "East"
(1,0)
```

2.2 getAllVectors (10 points)

Well, it should not be difficult to guess after PE1. Here is the function named `getAllVectors` will be implemented to convert a `list of Strings` representing a sequence of actions to a `list of Vectors`.

The signature of this function is below:

```
getAllVectors :: [String] -> [Vector]
```

SAMPLE I/O:

```
*PE2> getAllVectors ["North","East", "West","South"]
[(0,1),(1,0),(-1,0),(0,-1)]

*PE2> getAllVectors ["North","East","Stay","East","South"]
[(0,1),(1,0),(0,0),(1,0),(0,-1)]

*PE2> getAllVectors ["North","East","SouthEast","North","NorthWest"]
[(0,1),(1,0),(1,-1),(0,1),(-1,1)]
```

2.3 producePath (20 points)

You will implement a function named `producePath` which takes a `Point` and a `list of Strings` as a sequence of actions and returns a `list of Points` representing the path that the robot follows with the given sequence including its initial location.

Here is the signature of this function:

```
producePath :: Point -> [String] -> [Point]
```

It is guaranteed that initial coordinates will be always in the area but there is no limitation for the coordinates. So, negative values can be seen in the coordinates of a path.

SAMPLE I/O:

```
*PE2> producePath (0,0) ["North","East", "West", "South"]
[(0,0),(0,1),(1,1),(0,1),(0,0)]

*PE2> producePath (1,0) ["North","East", "West", "South","South","West"]
[(1,0),(1,1),(2,1),(1,1),(1,0),(1,-1),(0,-1)]

*PE2> producePath (0,0) ["North","East", "West", "South","South","West", "North","East","East","
    NorthEast"]
[(0,0),(0,1),(1,1),(0,1),(0,0),(0,-1),(-1,-1),(-1,0),(0,0),(1,0),(2,1)]
```

3 Part II (55 points)

3.1 takePathInArea (15 points)

Although there is no limitation for the robot's location, we would like to guide it in a specific area to explore things as mentioned in Section 1.1. Hence you will implement a function called `takePathInArea`, which takes a `list of Points` as the path, and `Dimensions` for the sizes of the area and returns a `list of Points` as the path **upto** first exceeding the area, if there is any. In other words, if the robot leaves the area and returns to the area back, the path after return is not included in the resulting list.

The signature of this function is below:

```
takePathInArea :: [Point] -> Dimensions -> [Point]
```

SAMPLE I/O:

```
*PE2> takePathInArea [(0,0),(0,1),(1,1),(0,1),(0,0)] (2,2)
[(0,0),(0,1),(1,1),(0,1),(0,0)]

*PE2> takePathInArea [(0,0),(0,1),(0,1),(1,1),(0,1),(0,0),(0,-1),(-1,-1)] (2,2)
[(0,0),(0,1),(0,1),(1,1),(0,1),(0,0)]

*PE2> takePathInArea [(0,1),(1,1),(1,1),(0,1),(0,0),(0,-1),(-1,-1),(-1,0),(0,0),(1,0),(2,1)] (2,2)
[(0,1),(1,1),(1,1),(0,1),(0,0)]

*PE2> takePathInArea [(1,0),(0,1),(1,1),(0,1),(0,2),(0,3),(0,2),(0,1),(0,0),(0,-1),(-1,-1),(-1,0)
    ,(0,0),(1,0),(2,1)] (2,2)
[(1,0),(0,1),(1,1),(0,1)]
```

3.2 remainingObjects (20 points)

As we got all we need now, we can focus on more goal-oriented functions. Suppose that there are objects in the some locations in the area we want to explore. If the robot visits the location having an object we assume that the object is picked up. Also, it is guaranteed that a location may have one object at most.

You will implement a function named `remainingObjects` to determine the objects that are not picked up by the robot. This function takes a `list of Points` as the path of the robot, `Dimensions` for the sizes of the area, and a `list of Points` representing the locations of the objects. The function returns a `list of Points` for the locations of the remaining objects in the area.

Here is the signature for this function:

```
remainingObjects :: [Point] -> Dimensions -> [Point] -> [Point]
```

If the robot leaves the area, it is **no more** possible to pick up objects even if the robot returns to the area.

SAMPLE I/O:

```
*PE2> remainingObjects  [(0,0),(0,1),(1,1),(0,1),(0,0),(0,-1),(-1,-1)] (2,2) [(0,0),(0,1),(1,0)]
[(1,0)]

*PE2> remainingObjects  [(0,1),(1,1),(0,1),(0,0),(0,-1),(-1,-1),(-1,0),(0,0),(1,0),(2,1)] (3,3)
    [(0,1),(1,1),(1,0),(2,0),(2,1),(2,2)]
[(1,0),(2,0),(2,1),(2,2)]

*PE2> remainingObjects [(0,0),(0,1),(1,1),(0,1),(0,2),(0,3),(0,2),(0,1),(0,0),(0,-1),(-1,-1)
    ,(-1,0),(0,0),(1,0),(2,1)] (3,3) [(0,1),(1,1)]
[]
```

3.3 averageStepsInSuccess (20 points)

Hang in there, it is the last function for this programming exam. In this function called `averageStepsInSuccess`, you will calculate how many steps are taken in average for the successful paths of the robot. A path is successful **if and only if** the robot never leaves the area and all of the objects are picked up.

Here is the signature of this function:

```
averageStepsInSuccess :: [[Point]] -> Dimensions -> [Point] -> Double
```

As the signature implies, this function takes a `list of lists of Points` as the list of paths, Dimensions for the sizes of the area and a `list of Points` as the locations of the objects in the area. The function returns a `Double` for the average number of the steps in successful paths. You can assume that there will be **at least one** successful path in the given list.

For further clarification of the task, here is definition of the average that we are looking for:

$$average = \frac{total\ \#\ of\ steps\ in\ successful\ paths}{\#\ of\ successful\ paths}$$

It should not require an extra adjustment in the calculations, however, to avoid any ambiguity, let us say that you must count the `"Stay"` actions for the steps of a path as well.

Hint: It is time to remind you the `castIntToDouble` function given in the template.
Another Hint: Try to get the successful paths before directly trying to get the average. Once you get the successful paths, the rest is more trivial.

SAMPLE I/O:

```
*PE2> averageStepsInSuccess [[(0,0),(0,1),(1,1),(0,1),(0,2),(0,3),(0,2),(0,1),(0,0),(0,-1),(-1,-1)
    ,(-1,0),(0,0),(1,0),(2,1)],[(0,0),(0,1),(1,1),(0,1)],[(0,0),(0,1),(1,1)]] (3,3) [(0,0),(0,1)
    ,(1,1)]
3.5

*PE2> averageStepsInSuccess [[(1,1),(0,1),(0,0),(0,1),(0,2),(0,3),(0,2),(0,1),(0,0),(0,-1),(-1,-1)
    ,(-1,0),(0,0),(1,0),(2,1)],[(0,0),(0,1),(1,1),(0,1)],[(0,0),(0,1),(1,1),(1,0),(0,0)]] (3,3)
    [(0,0),(0,1),(1,1)]
4.5
```

4 Regulations

1. **Implementation and Submission:** The template file named "pe2.hs" is available in the Virtual Programming Lab (VPL) activity called "PE2" on `OdtuClass`. At this point, you have two options:

   - You can download the template file, complete the implementation and test it with the given sample I/O on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   The second one is recommended. However, if you're more comfortable with working on your local machine, feel free to do it. Just make sure that your implementation can be compiled and tested in the VPL environment after you submit it.

   There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

3. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

   Important Note: The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official testcases to determine your ***actual*** grade after the deadline.