Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2021-2022

Programming Exam 1

Due date: 01 April 2022, Friday, 23:59

1 Problem Definition

In this programming exam, you will use your Haskell skills to help a researcher in robotics by providing some basic calculations. You will implement 6 Haskell functions in 2 parts. The functions in a part are structured in an incremental manner but the parts are independent from each other.

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.
- Data.List module is already imported in case you prefer using its some functionalities on given lists. It is not mandatory to use them, but it may ease your job. Importing any other modules is not allowed for this exam.
- 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.

- All of the simulations will take place in 2D environment. Therefore, you are not expected to consider beyond. In fact, here is the type synonym we use for the points in the environment in Part I:

```
type Point = (Double, Double)
```

- Another type synonym for Part II is given below for the radio signals to which the robots are exposed to in four directions. This radio signal will help the robots find their way in the environment.

```
type Signal = (Double, Double, Double, Double)
```

- Empty list will **not** be used in any testcases. Hence, when you are required to implement a function that takes a list you do not have to consider this edge case.

2 Part I (55 points)

2.1 getDistance (15 points)

You will implement a function named `getDistance` which takes two `Point`s and calculates a `Double` representing Euclidean distance between the given points. In this way, we will able to get the distance between the base and a robot in the environment.

Although it's a quite popular metric, let's remember the definition of Euclidean distance in 2D, for the sake of completeness:

$$d = \sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2}$$

Here is the signature of this function:

```
getDistance :: Point -> Point -> Double
```

SAMPLE I/O:

```
*PE1> getDistance (1,1) (1,2)
1.0

*PE1> getDistance (1,1) (2,2)
1.41

*PE1> getDistance (1,1) (4,5)
5.0
```

2.2 findAllDistances (15 points)

As the name implies the function named `findAllDistances` will be implemented to calculate the distances between the base and all robots. This function takes a `Point` and a `list of Point` and returns a `list of Double`.
Here is the signature of this function:

```
findAllDistances :: Point -> [Point] -> [Double]
```

SAMPLE I/O:

```
*PE1> findAllDistances (1,1) [(1,1),(1,2),(4,5)]
[0.0,1.0,5.0]

*PE1> findAllDistances (1,1) [(1,3),(5,1),(13,6)]
[2.0,4.0,13.0]

*PE1> findAllDistances (1,1) [(1,3),(2,2),(4,6)]
[2.0,1.41,5.83]
```

2.3 findExtremes (25 points)

You will implement a function named `findExtremes` which takes a `Point` as the coordinates of the base and a `list of Point` representing the location of the robots. This function finds the nearest and farthest robots to the base and return their locations as a `tuple` consisting of 2 `Point`s.

Here is the signature of this function:

```
findExtremes :: Point -> [Point] -> (Point, Point)
```

If more than one robots have the same minimum or maximum value for the distance to the base, you will favor the robot comes first in the written order to break the tie.

SAMPLE I/O:

```
*PE1> findExtremes (1,1) [(1,1),(1,2),(4,5)]
((1.0,1.0),(4.0,5.0))

*PE1> findExtremes (1,1) [(5,1),(1,3),(13,6)]
((1.0,3.0),(13.0,6.0))

*PE1> findExtremes (1,1) [(5,1),(3,1),(1,3),(13,6),(6,13)]
((3.0,1.0),(13.0,6.0))

*PE1> findExtremes (1,1) [(5,1),(3,1),(4,3),(12,13), (13,6)]
((3.0,1.0),(12.0,13.0))
```

3 Part II (45 points)

3.1 getSingleAction (25 points)

Although each robot has its own policy to explore the environment, sometimes further directives are needed to make them take a specific action. In that case, we will use the radio signal to guide the robot. Despite being given from a single source, with its sensors in 4 direction the robot will interpret a signal as a tuple consisting of 4 `Double` values (see Section 1 to remember its definition). These values correspond to four directions, namely North, East, South, West.

Here are the rules to determine the next action according to the given signal:

- You will check the difference in opposite directions. So, in vertical case, the value for North and the value for South will try to eliminate each other. Same rule applies to horizontal case with values for East and West.
- If all values are eliminated then the robot needs to sense the signal better. Hence the next action will be `"Stay"`.
- If only horizontal values are eliminated then the next action will be `"North"` or `"South"` depending on which direction has greater value.
- Similarly, if only vertical values are eliminated then the next action will be `"East"` or `"West"` depending on which direction has greater value.

- If there are positive values for both horizontal and vertical cases, then the action will be one of these depending on the directions having greater values than their opposites:
  - `"NorthEast"`
  - `"NorthWest"`
  - `"SouthEast"`
  - `"SouthWest"`

Now that you have the information needed to calculate the next action we can check the signature of this function:

```
getSingleAction :: Signal -> String
```

As the signature implies the function takes a `Signal` and returns one of the 9 `Strings` explained above. 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:

```
*PE1> getSingleAction (1.5,2,2.5,2)
"South"

*PE1> getSingleAction (1,1,1,1)
"Stay"

*PE1> getSingleAction (1.5,1.2,1.1,1.3)
"NorthWest"
```

3.2 getAllActions (10 points)

Not surprisingly, the function named `getAllActions` will do the determination of the actions for all robots according to the given `list of Signals` and returns a `list of Strings`.

Here is the signature for this function:

```
getAllActions :: [Signal] -> [String]
```

SAMPLE I/O:

```
*PE1> getAllActions [(1.5,2,2.5,2),(2.5,3.2,2.5,2),(1,1,1,1),(1.5,1.2,1.1,1.3)]
["South","East","Stay","NorthWest"]

*PE1> getAllActions [(3.5,2,2.5,2),(3.5,3.2,2.5,2),(1,1,1,1),(1.5,1.2,1.1,1.3)]
["North","NorthEast","Stay","NorthWest"]

*PE1> getAllActions [(3.5,2,2.5,2),(3.5,3.2,2.5,2),(5,1.2,1.1,1.4),(5.5,1.3,1.1,1.3)]
["North","NorthEast","NorthWest","North"]
```

3.3 numberOfGivenAction (10 points)

As the last function of the exam, `numberOfGivenAction` will calculate how many robots will take the given action according to given `list of Signals`.

Here is the signature of this function:

```
numberOfGivenAction :: Num a => [Signal] -> String -> a
```

As the signature implies the function takes a `list of Signals` and a `String` and returns a number belonging the Num class. Do not overthink about the Num class, the function is eventually required to calculate an integer. However, in this way you will most probably implement the function in your own way more freely.

Hint: If your solution has any difficulties to comply with the Num class restriction, `fromIntegral` function may save you the trouble.

SAMPLE I/O:

```
*PE1> numberOfGivenAction  [(1.5,2,2.5,2),(2.5,3.2,2.5,2),(1,1,1,1),(1.5,1.2,1.1,1.3)] "Stay"
1

*PE1> numberOfGivenAction  [(1.5,2,2.5,2),(2.5,3.2,2.5,2),(1,1,1,1),(1.5,1.2,1.1,1.3)] "West"
0

*PE1> numberOfGivenAction  [(3.5,2,2.5,2),(3.5,3.2,2.5,2),(5,1.2,1.1,1.4),(5.5,1.3,1.1,1.3)]
    "NorthEast"
1

*PE1> numberOfGivenAction  [(3.5,2,2.5,2),(3.5,3.2,2.5,2),(5,1.2,1.1,1.4),(5.5,1.3,1.1,1.3)]
    "North"
2
```

4 Regulations

1. **Implementation and Submission:** The template file named "pe1.hs" is available in the Virtual Programming Lab (VPL) activity called "PE1" 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.