Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2020-2021

Programming Exam 3

Due date: 22 April 2021, Thursday, 23:59

1 Problem Definition

As the 3^{rd} programming exam, you will be guiding a rock sampling mission on *the Planet Mars*. There will be `Haskell` functions to be implemented in 2 parts. Each function will be independently evaluated. However, some functions are designed in an incremental manner. Therefore, using some of the requested functions to implement some others (*in/between parts*) may really ease your job.

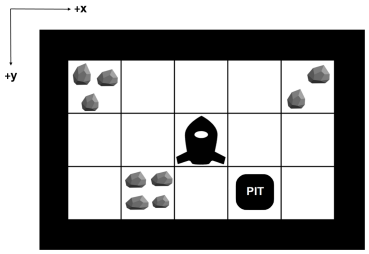

Figure 1: An Example Environment for The Mission

1.1 More Information about The Mission

Details for the mission and the environment are given in the following items (*Re-reading them after seeing the data types in Section 1.2 them is recommended*).

- The task includes robots which are responsible to collect some samples from the cells containing `Rocks` into the `SpaceCraft` in a grid-like 2D (M x N) environment as shown in Figure 1.
- No assumption should be done about the dimensions of the environment. M and N can be any integer greater than **0**.
- The environment can become severely challenging because it may have possible `Pits` where the robots may get stuck and lose energy without being able to move.
- If a cell does not include any of these objects, it will be a `Sand` cell by default.
- The coordinate of (0,0) is the top-left corner cell of the environment. Therefore, in the sample environment from Figure 1, (0,0), (4,0) and (1,2) are the cells including different amounts of `Rocks`. `SpaceCraft` is located at (2,1) and there is a `Pit` at (3,2) in this sample environment.

- x-coordinate will be decreasing while going `West` and increasing while going `East`. y-coordinate will be similarly affected by `North/South` actions. Be careful about handling these coordinates during the operation on `Grid` as a nested list of `Cell`.
- As you may have already noticed the robots are not actual parts of the environment according to the logic we use in our design. They can be at any `location` (*even in the same location, without any collision*) in this environment, however, they do not change the representation of the cells.
- The robots have six possible moves: going toward one of the four main directions as `North`, `East`, `South`, or `West`; `PickUp` one rock from a cell, and `PutDown` one rock from its storage to the spacecraft.
- The `energy` of a robot can be any integer in the range of **[0,100]** during the simulations.
- Each move of a robot will reduce its energy depending on the followings:
  - Going toward one of the main directions will decrease the energy by **1**.
  - `PutDown` action will decrease the energy by **3**.
  - `PickUp` action will decrease the energy by **5**.
- A robot must have the sufficient energy to complete an action.
- If a robot does not have the sufficient energy to take an action, its energy for the next step will be **0** without realization of the action effects over the environment and/or itself. For example, if the robot gets a `PickUp` action with the energy level of 3, it will fail to pick a rock from the cell into its storage. Namely, no change will occur in the environment and/or its storage, but its energy will be set to **0**.
- To be able to `PickUp` a rock, the robot must have less samples in its `storage` than its `capacity`. Otherwise, the robot will lose energy without being able to fetch the rock.
- `PickUp` and `PutDown` actions will not be used in an invalid location. In other words, a `PickUp` action will be given only for a `Rock` cell, and a `PutDown` action will be given in the cell containing `SpaceCraft`. However, if a `PickUp` action has been taken in a `Rock` cell with no sample left (`Rock 0`), the robot will redundantly lose energy.
- Another redundant energy loss occurs, if the robots try to move out of the environment. For example, if a robot takes a `North` action at the location (1,0), it will stay at the same location and its energy will be decreased by 1.
- Details regarding the functions by themselves will be separately covered in their subsections.

1.2 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.
- We will use the following data types in the functions. They are already given in the template file. You MUST them *as is*, no change on them is neither needed nor allowed.

```haskell
data Cell = SpaceCraft Int | Sand | Rock Int | Pit deriving (Eq,
   Read, Show)

type Grid = [[Cell]]
type Coordinate = (Int, Int)

data Move = North | East | South | West | PickUp | PutDown deriving
   (Eq, Read, Show)

data Robot = Robot { name :: String,
                     location :: Coordinate,
                     capacity :: Int,
                     energy :: Int,
                     storage :: Int } deriving (Read, Show)
```

2 Functions

2.1 Part I - Stuff to Warm-up

2.1.1 isInGrid (7.5 points)

This is the simplest function in this exam. You will write a function called `isInGrid` to determine whether a coordinate is within the borders of the given environment or not.

Here is the signature for this function:

```haskell
isInGrid :: Grid -> Coordinate -> Bool
```

SAMPLE I/O:

```
*PE3> isInGrid [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] (0,2)
True

*PE3> isInGrid [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] (5,2)
False

*PE3> isInGrid [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] (0,-2)
False
```

Note that the grid given in above samples is the same grid in Figure 1.

2.1.2 totalCount (10 points)

You will implement a function named `totalCount` having the signature below. This function will take a grid and compute the total number of rocks in the environment (*excluding the ones already gathered in SpaceCraft*). Obviously, if the environment does not have any rocks, it should return **0**.

```
totalCount :: Grid -> Int
```

SAMPLE I/O:

```
*PE3> totalCount [[Sand, Sand, Sand, Sand, Pit], [Sand, Sand, SpaceCraft
    3, Sand, Sand], [Sand, Pit, Sand, Pit, Sand]]
0

*PE3> totalCount [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]]
9

*PE3> totalCount [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 3, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]]
9
```

2.1.3 coordinatesOfPits (15 points)

The next function is `coordinatesOfPits` with the following signature:

```
coordinatesOfPits :: Grid -> [Coordinate]
```

As the name implies, this function will take a grid and return the list of `Coordinate`s for the cells containing a `Pit`. In the resulting list, the coordinates have to be ordered (*in the increasing order primarily with respect to x, then y*), as shown in the samples below.

SAMPLE I/O:

```
*PE3> coordinatesOfPits [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand
   , SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Sand, Sand]]
[]

*PE3> coordinatesOfPits [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand
   , SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]]
[(3,2)]

*PE3> coordinatesOfPits [[Rock 3, Sand, Sand, Sand, Pit], [Sand, Pit,
   SpaceCraft 3, Sand, Sand], [Sand, Pit, Sand, Pit, Sand]]
[(1,1),(1,2),(3,2),(4,0)]
```

2.1.4 tracePath (17.5 points)

You will implement a function called `tracePath` which will take a grid and a robot then produce the path of a robot in simulation (*according to the rules given in Section 1.1*) as a list of `Coordinate`s.

Here is the signature of this function:

```
tracePath :: Grid -> Robot -> [Move] -> [Coordinate]
```

Obviously, `PickUp` and `PutDown` action has no effect on the `location` of the robot. Also note that the coordinate will stay same if the `energy` of a robot is not sufficient and/or if it is in a cell containing a `Pit`.

SAMPLE I/O:

```
*PE3> tracePath [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] Robot {
   name="Fetchy", location=(2,1), capacity=10, energy=100, storage=0} [
   West, West, North, PickUp, South, East, East, PutDown]
[(1,1),(0,1),(0,0),(0,0),(0,1),(1,1),(2,1),(2,1)]

*PE3> tracePath [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] Robot {
   name="Fetchy", location=(2,1), capacity=10, energy=3, storage=0} [
   West, West, North, PickUp, South, East, East, PutDown]
[(1,1),(0,1),(0,0),(0,0),(0,0),(0,0),(0,0),(0,0)]

*PE3> tracePath [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] Robot {
   name="Fetchy", location=(2,1), capacity=10, energy=100, storage=0} [
   East, South, North, North, South, East, East, West]
[(3,1),(3,2),(3,2),(3,2),(3,2),(3,2),(3,2),(3,2)]
```

2.2 Part II - The Real Deal

2.2.1 energiseRobots (20 points)

You will implement a function named `energiseRobots` which simulates the wireless charging functionality of the `SpaceCraft`. When this functionality is triggered by calling `energiseRobots` with the given list of `Robots` in the environment, each robot in the range of SpaceCraft will gain energy depending on how far they are in the terms of *Manhattan Distance*. The function will return the list of updated `Robot`.

Additional energy for a robot will be calculated according to this formula:

$$gain = max(0, 100 - (|r_x - s_x| + |r_y - x_y|) \times 20), \tag{1}$$

where (r_x, r_y) and (s_x, s_y) are the coordinates of the robot and `SpaceCraft` respectively. While using this formula, do not forget that the energy of a robot CANNOT exceed 100.

Here is the signature of this function:

```
energiseRobots :: Grid -> [Robot] -> [Robot]
```

SAMPLE I/O:

```
*PE3> energiseRobots [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] [Robot {
   name="Fetchy", location=(1,1), capacity=10, energy=0, storage=0}]
[Robot {name = "Fetchy", location = (1,1), capacity = 10, energy = 80,
   storage = 0}]

*PE3> energiseRobots [[SpaceCraft 3, Sand, Sand, Sand, Rock 2], [Sand,
   Sand, Rock 3, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] [Robot {
   name="Balboa", location=(0,2), capacity=10, energy=3, storage=1},
   Robot {name="Fetchy", location=(4,2), capacity=10, energy=0, storage
   =0}]
[Robot {name = "Balboa", location = (0,2), capacity = 10, energy = 63,
   storage = 1},Robot {name = "Fetchy", location = (4,2), capacity = 10,
    energy = 0, storage = 0}]
```

2.2.2 applyMoves (30 points)

The last function is `applyMoves` which is very similar to `tracePath` function regarding its logic. However, this time you will have to keep track of all action effects on both the environment and the robot itself. This function returns the pair of the grid and the robot as a tuple representing their status after the simulation.

Here is the complete signature:

```
applyMoves :: Grid -> Robot -> [Move] -> (Grid, Robot)
```

SAMPLE I/O:

```
*PE3> applyMoves [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] Robot {
   name="Fetchy", location=(2,1), capacity=10, energy=100, storage=0} [
   West, West, North, PickUp, South, East, East, PutDown]
([[Rock 2,Sand,Sand,Sand,Rock 2],[Sand,Sand,SpaceCraft 1,Sand,Sand],[
   Sand,Rock 4,Sand,Pit,Sand]],Robot {name = "Fetchy", location = (2,1),
    capacity = 10, energy = 86, storage = 0})

*PE3> applyMoves [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] Robot {
   name="Fetchy", location=(2,1), capacity=10, energy=3, storage=0} [
   West, West, North, PickUp, South, East, East, PutDown]
([[Rock 3,Sand,Sand,Sand,Rock 2],[Sand,Sand,SpaceCraft 0,Sand,Sand],[
   Sand,Rock 4,Sand,Pit,Sand]],Robot {name = "Fetchy", location = (0,0),
    capacity = 10, energy = 0, storage = 0})

*PE3> applyMoves [[Rock 3, Sand, Sand, Sand, Rock 2], [Sand, Sand,
   SpaceCraft 0, Sand, Sand], [Sand, Rock 4, Sand, Pit, Sand]] Robot {
   name="Fetchy", location=(2,1), capacity=10, energy=100, storage=0} [
   East, South, North, North, South, East, East, West]
([[Rock 3,Sand,Sand,Sand,Rock 2],[Sand,Sand,SpaceCraft 0,Sand,Sand],[
   Sand,Rock 4,Sand,Pit,Sand]],Robot {name = "Fetchy", location = (3,2),
    capacity = 10, energy = 92, storage = 0})
```

3 Regulations

1. **Implementation and Submission:** The template file named "pe3.hs" is available in the Virtual Programming Lab (VPL) activity called "PE3" 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.

   There will be a reasonable timeout to eliminate erroneous codes as before PEs, however, the performance of your code is not evaluated. That is, do not worry about the complexity as long as your code stays responsive.

   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.