Middle East Technical University

Department of Computer Engineering

CENG 242

Programming Language Concepts
Spring 2023
Programming Exam 1

Due: 2 April 2023 23:55
Submission: **via ODTUClass**

1 Problem Definition

In this programming exam, you will be using Haskell to construct the building blocks of a recipe calculator system, which keeps track of recipes, ingredients and generates shopping lists for convenient management. To this end, you will be implementing 5 Haskell functions.

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 needed.
- 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.

- Data types describing recipes and price listings of ingredents are already defined for you in the homework files, as given below:

```
data Recipe = Recipe String [(String, Double)] deriving Show
data Price = Price String Double Double deriving Show
```

  - A recipe (the `Recipe` type) has a name (`String`) and a list of ingredients and their quantities (`[(String, Double)]`). For example:

```
Recipe "Toast" [("Bread", 120), ("Cheese", 40), ("Butter", 10)]
```

    means that to make toast, 120gr of `Bread`, 40gr of `Cheese` and 10gr of `Butter` is required.
  - A price listing (the `Price` type) includes the name of the ingredient, a quantity, and the price for that amount of the ingredient, in that order. For example:

```
Price "Tomato" 1000 20
```

    means that it costs 20TL to buy 1kg (1000 gr) of tomatoes.

2 Functions

2.1 getIngredientCost (15 pts.)

This function takes an `(IngredientName, Quantity)` pair and a `[Prices]` (a list of prices), and calculates how much it costs to purchase that ingredient in the given amount. You may assume that the list of prices always include a price for the given ingredient. Please make sure that you don't forget to take `Quantity` into account.
This function has the signature:

```
getIngredientCost :: (String, Double) -> [Price] -> Double
```

Here is an example call to the function:

```
ghci> getIngredientCost ("Pea", 500) [Price "Fig" 1000 30, Price "Pea" 1000 15]
7.5
```

2.2 recipeCost (15 pts.)

Here, you will implement a function that takes a `Recipe` and calculates how much it would cost to buy the ingredients for that recipe. The cost of a `Recipe` is simply the cost of all its ingredients, combined. Every ingredient of the recipe is always available in the price listings. Make sure that you take the ingredient amounts into account!
Here is the function signature:

```
recipeCost :: Recipe -> [Price] -> Double
```

And an example:

```
ghci> tea = Recipe "Tea" [("Water", 200), ("Tea", 5)]
ghci> prices = [Price "Tea" 100 50, Price "Coffee" 50 75, Price "Water" 2000 4]
ghci> recipeCost tea prices
2.9
```

2.3 missingIngredients (20 pts.)

This function takes a `Recipe` and a list of ingredients that the user already has in his pantry (as `(IngredientName, Quantity)` pairs) and returns a list containing how much of each ingredient is missing. If there is enough of an ingredient, then that ingredient should not appear in the list.
The function signature is:

```
missingIngredients :: Recipe -> [(String, Double)] -> [(String, Double)]
```

And it is used like so:

```
ghci> stock = [("Tea", 50), ("Water", 100)]
ghci> tea = Recipe "Tea" [("Water", 200), ("Tea", 5)]
ghci> missingIngredients tea stock
[("Water", 100)]
```

2.4 makeRecipe (20 pts.)

Similar to the previous function, this function takes a list of available ingredients and a recipe, but this function calculates how much of each ingredient would be left after making the recipe. If all ingredients of the recipe is not available in the given ingredients, then the recipe cannot be made, in such a case the amount of available ingredients should remain unchanged.
This function has the signature:

```
makeRecipe :: [(String, Double)] -> Recipe -> [(String, Double)]
```

It would be called as:

```
ghci> stock = [("Bread", 500), ("Olive", 250), ("Butter", 50), ("Cheese", 400)]
ghci> toast = Recipe "Toast" [("Bread", 120), ("Cheese", 40), ("Butter", 10)]
ghci> makeRecipe stock toast
[("Bread", 380), ("Olive", 250), ("Butter", 40), ("Cheese", 360)]
```

Another example call:

```
ghci> stock = [("Bread", 70), ("Olive", 250), ("Butter", 50), ("Cheese", 400)]
ghci> toast = Recipe "Toast" [("Bread", 120), ("Cheese", 40), ("Butter", 10)]
ghci> makeRecipe stock toast
[("Bread", 70), ("Olive", 250), ("Butter", 50), ("Cheese", 400)]
```

2.5 makeShoppingList (30 pts.)

The final function of this Programming Exam, `makeShoppingList` makes a shopping list for the user when given the list of ingredients that the user already has, a list of recipes that they wish to cook, and the list of prices (which, again, is guaranteed to include any ingredient needed) of the store at which they will shop. It gives back a list that contains every ingredient that the user will need, along with its amount. It also includes how much each ingredient will cost, so that the user knows how much money they will be spending.
Here is its function signature:

```
makeShoppingList :: [(String, Double)] -> [Recipe] -> [Price] -> [(String, Double, Double)]
```

And it can be called like:

```
stock = [("Bread", 400), ("Cheese", 20)]
prices = [Price "Egg" 60 1, Price "Cheese" 100 10
          Price "Butter" 100 25, Price "Bread" 250 5]
recipes = [Recipe "Fried Egg" [("Egg", 60), ("Butter", 20)],
           Recipe "Toast" [("Bread", 120), ("Cheese", 40), ("Butter", 10)]]

ghci> makeShoppingList stock recipes prices
[("Egg", 60, 1), ("Cheese", 20, 2), ("Butter", 30, 7.5)]
```

3 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 (and also your own test cases) 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.

   If you work on your own machine, 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 test cases to determine your ***actual*** grade after the deadline.