M21274 – MATHFUN
Discrete Mathematics and Functional Programming

Worksheet 2: Introduction to Functional Programming II

Introduction

The programming exercises give you practice using guards and local definitions (using he `where` keyword). The written exercises will help you develop an understanding of how expressions are evaluated in Haskell (but these will not be used for the sign-off). Make sure your solutions to the programming exercises are in a file called `Week2.hs`, test them, and show them to us for feedback/sign-off in Week 3/4.

Question: As part of a heart rate monitoring system on a smartwatch, write a function `heartMonitor` that takes the person's age and their resting heart beats per minute (BPM). The function should then output a warning message if the BPM is above the maximum value specified in the following table, or a "normal heart heart" message if the BPM is OK.

Age	Max resting BPM
0-20	170
21-40	155
41-60	140
61-80	130
81+	100

Since we take two integers and return a string, the signature of `heartMonitor` is:

```
heartMonitor :: Int -> Int -> String
```

Solution: Below is our first attempt; write it inside `Week2.hs` and save the file. Notice that every guard (lines beginning with the pipe symbol "|") is indented:

```
heartMonitor :: Int -> Int -> String
heartMonitor age bpm
  | age > 80 && bpm > 100 = "High heart rate for 81+!"
  | age > 60 && age <= 80 && bpm > 130 = "High heart rate for 61-80!"
  | age > 40 && age <= 60 && bpm > 140 = "High heart rate for 41-60!"
  | age > 20 && age <= 40 && bpm > 155 = "High heart rate for 21-40!"
  | age >= 0 && age <= 20 && bpm > 170 = "High heart rate for 0-20!"
  | otherwise = "Normal heart rate"
```

Now load it in GHCi and test it by evaluating expressions such as:

```
heartMonitor 20 150
```

We can improve our code quality by removing unnecessary checks. Remove all checks on the upperbound of the ages (red highlights show the positions of the removed checks):

```
heartMonitor :: Int -> Int -> String
heartMonitor age bpm
  | age > 80 && bpm > 100 = "High heart rate for 81+!"
  | age > 60 &&  bpm > 130 = "High heart rate for 61-80!"
  | age > 40 &&  bpm > 140 = "High heart rate for 41-60!"
  | age > 20 &&  bpm > 155 = "High heart rate for 21-40!"
```

```
  | age >= 0 && bpm > 170 = "High heart rate for 0-20!"
  | otherwise = "Normal heart rate"
```

Worked example 2

Question: The calories of a pizza are made up of the calories in its base and its toppings. Every pizza base has 11.5 calories per square centimetre and the calories in the available toppings are provided in the table below. We will assume for simplicity that the topping covers the entire base of the pizza.

We wish to write a function `pizzaCalories` where the diameter (in cm) is given as an `Int` and the topping as a `String`; the function should return the number of calories as a `Float`.

Topping	Calories per cm^2
"veggie"	2.5
"tuna"	4
"pepperoni"	6

Solution: Let us start with a naïve, solution. Add the following at the end of `Week2.hs`. Note that we are using `fromIntegral` so that we can divide the diameter, which is an `Int`, by 2:

```
pizzaCalories :: Int -> String -> Float
pizzaCalories diameter toppings
  | toppings == "pepperoni" = (11.5 + 6) * pi * (fromIntegral diameter / 2) ^ 2
  | toppings == "tuna" = (11.5 + 4) * pi * (fromIntegral diameter / 2) ^ 2
  | toppings == "veggie" = (11.5 + 2.5) * pi * (fromIntegral diameter / 2) ^ 2
  | otherwise = 11.5 * pi * (fromIntegral diameter / 2) ^ 2
```

Notice how we are calculating the area of the pizza from the diameter in every case. To avoid this repetition, we can create the local definition `area` using the `where` keyword. Notice that the parameters `diameter` and `topping` are accessible in the `where` clause:

```
pizzaCalories :: Int -> String -> Float
pizzaCalories diameter toppings
  | toppings == "pepperoni" = (11.5 + 6) * area
  | toppings == "tuna" = (11.5 + 4) * area
  | toppings == "veggie" = (11.5 + 2.5) * area
  | otherwise = 11.5 * area
  where
    area = pi * (fromIntegral diameter / 2) ^ 2
```

Just like functions, local definitions can have guards. An alternative definition of the `toppingCalories` function,using the `where` keyword can therefore be given as:

```
pizzaCalories :: Int -> String -> Float
pizzaCalories diameter toppings = (11.5 + toppingCalories) * area
  where
    area = pi * (fromIntegral diameter / 2) ^ 2
    toppingCalories
      | toppings == "pepperoni" = 6
      | toppings == "tuna" = 4
      | toppings == "veggie" = 2.5
      | otherwise = 0
```

Programming exercises

For each of the questions in this section, write a new function inside `Week2.hs`. Make sure to test each function using the provided table (and other appropriate data) before moving onto the next exercise.

1. Write a new version of the `absolute` function that you wrote as part of worksheet 1 (question 10). This time, use guards instead of `if then else`.

2. Using guards, write a function `sign` that returns 1 for positive, -1 for negative and 0 for zero-valued arguments as shown in the table:

```
sign :: Int -> Int
```

Test	Output
`sign 5`	`1`
`sign 0`	`0`
`sign (-5)`	`-1`

3. Write a function `howManyEqual` that checks how many of its 3 arguments are equal.

```
howManyEqual :: Int -> Int -> Int -> Int
```

Test	Output
`howManyEqual 3 2 1`	`0`
`howManyEqual 3 2 3`	`2`

4. Given a square and the length of one of its sides, the length of its diagonal can be calculated using the following formula:

$$diagonalLength = \sqrt{2 \times sideLength^2}$$

   Write a function `sumDiagonalLengths` that takes the side length of three different squares as its arguments. It should calculate the length of the each diagonal using the above formula and return the sum of these values.

```
sumDiagonalLengths :: Float -> Float -> Float -> Float
```

Test	Output
`sumDiagonalLengths 4 10 16`	`42.426407`
`sumDiagonalLengths 8 29 46`	`117.37973`

   Avoid repetition by using the `where` clause in your solution.

5. A taxi company calculates fares based on the distance travelled. Fares start at £2.20. 50p is added for each kilometre covered for the first 10 kilometres; and 30p is added for each additional kilometre. Write a function `taxiFare` that takes the distance in kilometres and returns the fare in pounds.

```
taxiFare :: Int -> Float
```

Test	Output
`taxiFare 4`	`4.2`
`taxiFare 10`	`7.2`
`taxiFare 14`	`8.4`

6. Write a function `howManyAboveAverage` that takes three integers as its arguments and returns how many of them are above the average.

```
howManyAboveAverage :: Int -> Int -> Int -> Int
```

   Hint: We have not provided the test cases, but note that there are 3 possible return values. Work out what these cases are before writing your solution.

7. Write a function `validDate` that takes integers representing a day and a month. Then it should return `True` if and only if the date is valid.

```
validDate :: Int -> Int -> Bool
```

Test	Output
`validDate 29 3`	`True`
`validDate 31 4`	`False`

   Note: The first argument is the day, and the second one is the month. Also, assume that February always has 28 days.

8. Assuming that all years divisible by 4 are leap years (29 days in February), write a function `daysInMonth` which returns the number of days in a given month and year.

```
daysInMonth :: Int -> Int -> Int
```

Test	Output
`daysInMonth 2 2020`	`29`
`daysInMonth 2 2022`	`28`
`daysInMonth 1 2023`	`31`

   Note: Assume that the given values for the year and month supplied to your function are always valid. We also recommend using `where` to simplify your solution.

Written exercises

Check out the "Evaluation and calculation" lecture notes for an explanation. Write your solution as a comment inside `Week2.hs` (see below) or in a separate text file (`Week2.txt`).

```
{-
This is a multi-line comment
-}
```

1. For your `sumThree` function from worksheet 1, write down the calculations that evaluate the following expressions:

   - `sumThree 3 5 7`

- `sumThree 8 (1 + 3) 2`

2. For your `threeDifferent` function from worksheet 1, write the calculations that evaluate the following expressions:
   - `threeDifferent 1 4 2`
   - `threeDifferent 1 7 7`
3. For your `howManyEqual` function from this worksheet, write the calculations that evaluate the following expressions:
   - `howManyEqual 3 5 2`
   - `howManyEqual 5 2 5`