Functional Programming Assignment Report

Neel Mittal - 150101042
Prashansi Kamdar 150101047

I. Introduction

THIS report summarizes the solutions developed to the given problems, and answers the questions asked.

II. Impure Functions in Haskell

Functions written entirely in Haskell are pure. Functions that deal with IO are functions which interact with the the world outside of Haskell and thus produce a change which makes them impure. Functions with IO in the signature would be impure functions as they deal with IO. Since none of the functions we write have IO in their signature, none of our functions are impure.

III. Palindrome Maker

The following functions were used to solve this problem :

- **ord** : This function returns the ascii code for a character.
- **abs** : This function returns the absolute value of the given integer.
- **dubchanges** : This function returns the number of moves required to match two given strings as per the given operation.
- **getchanges** : This function gets the number of required to make the input string a palindrome.
- **div** : This function divides two given integers and returns the quotient.
- **reverse** : This function reverse the input string.

Thus in total, 6 functions were used, only **dubchanges** and **getchanges** are defined by us. All are pure.

IV. Free Coupon Problem

The following functions were used to solve this problem :

- **mod** : This function returns the remainder after the division between the given two integers.
- **spendToken** : This function returns the number of parathas that can be bought by spending the given amount of tokens and the given cost.
- **countParathas** : This function returns the number of parathas that can be bought given initial money and cost of a paratha in terms of money and tokens.
- **div** : This function divides two given integers and returns the quotient.

Thus in total, 4 functions were used, only **spendToken** and **countParathas** are defined by us. All are pure.

V. Minimum Number of Moves

In order to get the minimum number of moves, we should always choose the one with the highest salary currently, as that would cause the absolute difference between the highest and lowest salaries to reduce. If the person with the highest salary is not picked, then the gap between the highest and lowest salaries remains the same and more moves would be required. Thus, picking the highest salaried person and increasing everyone's salary besides him can be thought of as reducing the salary of this person by 1. Therefore, the number of moves would just be the number of moves required to reduce the salary of each person to the minimum salary at the beginning as at some point during the moves, a person whose salary is not the minimum at the beginning, would become the person with the highest salary.

The following functions were used to solve this problem :

- **sum** : This function returns the sum of the integers present in the given list of integers.
- **getMoves** : This function returns the number of moves required to make all the salaries equal.
- **length** : This function returns the length of a given list.
- **minimum** : This function returns the minimum element in a list.

Thus in total, 4 functions were used, only **getMoves** is defined by us. All are pure.

VI. House Planner

The algorithm used is as follows :

1) We keep all the possible dimensions of the rooms subject to the given constraints as *allDims*
2) Given the area and number of bedrooms and halls required, we first devise the number of kitchens and bathrooms.
3) Once we have the required number of all the rooms, we then filter out the possible dimensions from *allDims* by checking the area occupied if a particular set of dimensions is used.
4) From all the possible dimensions, we pick the one occupying the most area, and return this as the answer

We only list the functions we have written to solve this problem to keep the report brief :

- **getKitchen** : This function returns the number of kitchens required given the number of bedrooms. This function can be modified to reflect change in constraint.
- **getBathroom** : This function returns the number of bathrooms given the number of bedrooms. This function can be modified to reflect change in constraint.

- **multiAll** : This function takes an Integer and a list of integers, and return the product of the Integer and the first two Integers in the list.
- **addInfo** : This function takes the area and the required number of rooms and returns the possible set of dimensions satisfying all of the given criteria except the criteria to optimize unused space.
- **getAll** : This function returns all the possible dimensions given the area and required bedroom and hall quantity.
- **getLowest** : This function gets the minimum value at the 0^{th} index of a list of list of list of Integers.
- **getMin** : This function returns the set of possible dimensions which have the best unused space.
- **getBest** : A wrapper around **getMin** for easier readability.
- **design** : A function the returns the expected string given the maximum area allowed, and the requirements for the number of bedrooms and halls.

Thus, in total, 9 new functions have been used and all of these are pure.

VII. Benefit of Lazy Evaluation

Lazy Evaluation means that expressions are not evaluated until their return values are needed. It can be used to improve performance of the House Planner problem since the set of possible dimensions is large and storing it in memory would not be efficient. Haskell can just generate the required set of dimension and do the required processing to determine whether that set of dimensions is satisfies all the mentioned criteria, and once it is done with this, only then it will move on to the next set of dimensions. Thus in the case that the set of dimensions can't be stored in the memory, Haskell can still do the required work by using just enough memory for one set of dimensions. It also speeds up the code execution as at each step in the processing, the set of dimensions being processed can become invalid and can be discarded quickly, moving on to the next set of dimensions, and thus the huge set of dimensions will be processed more easily.

VIII. Advantage of Haskell

Since there is a lack of side effect, Haskell provides less chances of errors related to entering invalid states in cases of an imperative program. Since the functions output only depends on the input, debugging each individual function becomes easy as only the function being debugged needs to be examined, not the entire code. When dealing with functions having no side effects, once the function has been debugged, there is no probability of there ever being any new bug in that piece of code. Since the functions are pure, they have less dependencies that impure functions and make the code more reusable than impure functions written in the imperative programming paradigm.