Coverage & Logistics

This homework covers material through the first four chapters of *Haskell: The Craft of Functional Programming* (HCFP).

This homework is officially due in class on **Thursday, February 2**. However, it comes with an automatic extension: anything submitted to the CIS 252 bin near CST 4-226 by **noon on Friday, February 3** will be accepted as being on time.

You may work singly or in pairs on this assignment.

Grading Guidelines & Advice

Your submission will be graded in accordance with the following criteria:

1. Does your code work correctly? (If it doesn't work correctly, do you have a comment indicating its limitations?)
2. Is your code clear and easy to read? Code that is unnecessarily complicated or difficult to read will likely lose some points.
3. Do you have an adequate collection of test cases?
4. Did you follow instructions (fill out disclosure cover sheet correctly, generate a clean transcript, etc.)?

Work on one problem at a time: get it working before moving on to the next problem. (Writing all your code at once will likely result in a series of errors and a lot of frustration on your part. It is usually **much easier** to work in small increments.)

Generate your final transcript **only after all of your code works**. Take pity on the graders, and make sure that your solutions will be easy to grade/verify.

Exercises

1. Write a Haskell function

   ```
   compareChars :: Char -> Char -> Char -> String
   ```

   such that `(compareChars a b c)` returns a string indicating how many of the characters `a`, `b`, `c` are equal to one another:

   - If all three are the same, it returns `"All equal"`.
   - If exactly two of them are the same, it returns `"Two match"`.
   - If all three are different, it returns `"All distinct"`.

   For example, `(compareChars 'R' (toUpper 'r') 'R')` returns `"All equal"`, `(compareChars 'R' 'r' 'R')` returns `"Two match"`, and `(compareChars 'a' 'B' 'b')` returns `"All distinct"`.

2. Write a Haskell function

   ```
   combine :: Int -> Int -> Int -> Int
   ```

   with the following behavior:

   - When `x`, `y`, and `z` all correspond to digit values (i.e., integers between 0 and 9, inclusive), `combine x y z` returns the integer given by the sequence of digits `x y z`.

     (That is, `x` is treated as the digit in the hundreds place, `y` is treated as the digit in the tens place, and `z` is treated as the digit in the ones place.)
   - In all other cases, the function returns `-1`.

   For example, `combine 4 2 8` returns `428`, `combine 0 3 0` returns `30` (because $030 = 30$), and `combine 4 12 3` returns `-1`.

3. Write a Haskell function

   ```
   splitFloat :: Float -> (Integer, Float)
   ```

   such that `splitFloat num` "splits" `num` into its whole-number part and its fractional (i.e., to the right of the decimal point) part; when `num` is negative, only the integer portion should contain the negative sign. For example, `(splitFloat 3.78)` should return `(3,0.78)`, and `splitFloat (-3.78)` should return `(-3,0.78)`. *Note: to create a pair result, simply use parentheses and commas in the obvious way: for example, if `x` has the value 10, then `(x+1,x-3)` evaluates to the pair `(11,7)`.*

   Two hints: (1) Use some of the following built-in Haskell functions to convert between `Integer` and `Float`: `ceiling`, `floor`, `round`, `fromInteger`. (2) Figure out how to handle positive numbers first; once that is working, determine how to adjust your solution for negative numbers.

4. A small smartphone company offers two different pricing plans for its wireless customers (all prices below represent monthly costs):

   - Under the **standard-pricing plan**, customers pay a base price of \$50 per month and receive unlimited talk, unlimited texting, and 4 gigabytes (GB) of data usage for use on one device. They may add additional devices for \$10 each, and each additional GB of data usage costs \$15.
   - Under the **power-user plan**, customers pay a base cost of \$80 per month for one device. They may add additional devices for \$8 each per month. They receive unlimited talk, unlimited texting, and 10 gigabytes (GB) of data usage *plus and additional 2 GB for each device*. Each additional GB of data usage beyond that allowance costs \$20.

   In addition, the company offers a one-time discount of \$25 to new customers. The following scenarios illustrate how this fee structure works:

   - Amy uses the standard-pricing plan and has two devices associated with her account. If she uses 7GB in a particular month, her cost is \$105: \$50 for the base fee, \$10 for her second device, and $\$15 \times 3$ for the data overage.

     If Amy is a new customer, she pays \$80.
   - Brad uses the power-user plan and has three devices associated with his account: he is thus entitled to 16 GB each month (that is, 10GB + $3 \times$ 2GB). If he uses 20GB in a particular month, his cost is \$176: \$80 for the base fee, $\$8 \times 2$ for the two additional devices, and $\$20 \times 4$ for the data overage.

     If Brad is a new customer, he pays \$151.
   - Corey also uses the power-user plan and has two devices associated with her account. If she uses 7GB in a particular month, her cost is \$88: \$80 for the base fee and \$8 for the additional device (she's within her data allowance). If Corey is a new customer, she pays \$63.

   (a) Write a Haskell function

   ```
   stdCost :: Int -> Int -> Bool -> Int
   ```

   such that `(stdCost dev gb new)` calculates the monthly cost (in dollars) under the standard pricing plan for a customer with `dev` devices who uses `gb` gigabytes of data over the course of the month; the Boolean `new` is true precisely when the customer is a new customer. If `dev` is less than 1, or if `gb` is negative, your function should return `-1`.

   For example, `stdCost 2 7 False` should return `105`, and `stdCost 2 7 True` returns `85`.

   (b) Write a Haskell function

   ```
   powerCost :: Int -> Int -> Bool -> Int
   ```

   such that `(powerCost dev gb new)` calculates the monthly fee under the power-user plan for a customer with `dev` devices who uses `gb` gigabytes of data over the course of the month; the Boolean `new` is true precisely when the customer is a new customer. If `dev` is less than 1, or if `gb` is negative, your function should return `-1`.

   For example, `powerCost 3 20 False` should return `176`, and `powerCost 3 20 True` should return `151`.

   (c) Write a Haskell function

   ```
   bestPlan :: Int -> Int -> String
   ```

   such that `(bestPlan dev gb)` determines which pricing plan is the better choice (i.e., less expensive) for an established (i.e., not new) customer with `dev` devices who expects to use `gb` gigabytes of data in a given month. It returns its answer as a string: `"Standard"`, `"Power User"`, or `"Same cost"`.

   For example, `(bestPlan 1 3)` should return `"Standard"`, whereas `(bestPlan 2 7)` should return `"Power User"`.

What to turn in

Turn in hard copies of (1) your source code and (2) a clean transcript demonstrating convincingly that your code is correct. As always, include a disclosure cover sheet.

Note: To provide a *convincing demonstration* of correctness, include test cases that handle the various input possibilities your code may encounter.