CPSC 312
Assignment #4 - Haskelling
Due no later than midnight of Friday, November 20th, 2015.
Please include your official full name, your student id as well as your ugrad id in your submission.

Submission:
There are four questions in this homework.
Put your solutions to all questions into one **.hs** file and put your identification there as well.
Any code/text that is not Haskell should be commented out. Any Haskell code that does not work should also be commented out with an added explanation of why it doesn't work.
All your Haskell functions must be accompanied by explicit type declaration.
Functions missing type declaration receive only partial grade.
All your Haskell functions must be accompanied by comments that explain their purpose and behaviour. Code with no comments receives NO grade.
Use handin for submission with the course name **cs312** and assignment name **assign4**.

Important: Collaboration Policy
For this assignment you may work with **one** other current CPSC312 student, in which case you will **submit one solution** together (which includes identifications of both students). You may not however, work with two or more other students. You also don't have the option of working "primarily" with one student but "consulting" with another occasionally. A team of two is exactly that, not a team of 2 1/2.

Two students who work together are equally responsible for all the work submitted. Do not split the assignment such that one student does some of the problems and the other student does the other problems. That won't get you where you want to be at exam time. We recommend these often-repeated pair-programming techniques (borrowed from someone, who borrowed them from someone else, and so on):

Both programmers share a computer, a monitor, and a keyboard.

One programmer is the "pilot" for a while and types code, while the other programmer is the "navigator", watching what is being typed and critiquing it.
The programmers switch roles at regular intervals.
Again, both programmers jointly own everything that was typed, no matter who typed it.

There is no grade penalty for working with **one** other current CPSC 312 student, but you must acknowledge the collaborative effort by putting the names of both students on the work submitted for evaluation.

You may also choose to not engage in collaborative learning. In this case, you will work entirely on your own. You don't get extra marks or an adjusted workload for working by yourself. And please keep in mind that working alone means working alone, it does not mean you work "primarily" alone while you "consult" with another student. You either work as a pair or solo; there is no middle ground.

Problem 1 : Functional Design (10 points)

(This problem is stolen from the book Geek Logik by Garth Sundem.)

Should you eat something scary from the back of the fridge, or just order Chinese again? With no outside influences, we would order Chinese food every night, because Chinese food is good – especially Chinese food that is either fried or covered in enough sweet sticky sauce to technically make this sauce the entree and the food part the garnish. However, your monthly food budget might not support a nightly $18.95 meal, and eventually your clogged arteries will rebel...

The equation below balances your stress level and the likelihood of the National Science Institute studying your fridge for clues to the origin of life against your budget and current cholesterol level. Also, if the Wok and Talk down the street has started recognizing your number on caller ID and now answers your phone call with a personal greeting, this intrusive intimacy may lead you to explore other cuisine options (or at least pick a new Chinese place where the busboys don't consider you a shut in). This equation helps you determine the value for K, your current Kung Pao factor. If K is greater than 1, you should order Chinese take out.

$$K = \left(\frac{N}{30} - \frac{D_s}{D_m} \right) + \frac{10\, S^2 \sqrt{R}}{C\,(F_t - F_f + 1)}$$

where

R = the number of seconds of phone time that it takes for the person at the restaurant to recognize your voice (maximum of 30 seconds)

D_m = your monthly food budget in dollars

D_s = the amount of money in dollars that you have already spent this month on food

N = number of today's date (e.g., if today is September 15, then N = 15)

C = your current blood cholesterol level (140-200 is the normal range; 300 is very high)

F_t = total number of substantial food items currently in your refrigerator (mustard and baking soda don't count)

F_f = number of those food items that have at least a 20 percent chance of being either furry or living

S = how stressed out are you (1-10 with 10 being "tight as a banjo string")

Your task is to write a Haskell program called kungPaoFactor that, given values for the eight parameters listed above, computes the corresponding Kung Pao Factor K. Here's how the parameters must be ordered in the top-level procedure call:

```
kungPaoFactor r dm ds n c ft ff s
```

Again, don't forget to use abstraction to simplify things for yourself and your TAs.

You shouldn't need any built-in functions other than basic arithmetics for this function. The power function that operates with real numbers is (**) in Haskell, just in case you needed it.

Problem 2 : Recursion (10 points)

The sum of the first n terms of the harmonic sequence is defined as

1 + 1/2 + 1/3 + 1/4 + 1/5 + ... + 1/n

a. Using Haskell, construct the function **harmonic** which takes one argument, an integer n which is greater than 0, and returns the sum of the first n terms of the harmonic sequence as a floating point number. You can use Haskell's built-in arithmetic operations; you don't have to use functions you've written as solutions to other problems here. Your solution should use **natural recursion** (i.e., not tail recursion) to compute the result.
b. Construct the function **harmonic_tr** which takes one argument, an integer n which is greater than 0, and returns the sum of the first n terms of the harmonic sequence as a real number. You may use Haskell's built-in arithmetic operations; you don't have to use functions you've written as solutions to previous problems. Your solution should use **tail recursion** to compute the result.

You may (or may not) also need this type conversion function:

```
fromIntegral :: (Integral a, Num b) => a -> b
```

Look in Prelude for other functions from lecture.

Problem 3: List Recursion (20 points)

For each of the following problems, provide two (recursive) solutions: one using pattern matching and the other not using pattern matching. The only built-in functions allowed in this question are **':', 'head', 'tail', 'null',** and **'elem'.**

Please add _pm to the end of the function names for pattern-matching based implementations.

Here is an example:

```
-- without pattern matching
myappend list1 list2
  | list1 == [] = list2
  | otherwise   = (head list1):(myappend (tail list1) list2)

-- with pattern matching
myappend_pm [] list2     = list2
myappend_pm (x:xs) list2 = x:(myappend_pm xs list2)
```

1) myremoveduplicates

myremoveduplicates "abacad" => "bcad"
myremoveduplicates [3,2,1,3,2,2,1,1] => [3,2,1]

2) mynthtail

mynthtail 0 "abcd" => "abcd"
mynthtail 1 "abcd" => "bcd"
mynthtail 2 "abcd" => "cd"
mynthtail 3 "abcd" => "d"
mynthtail 4 "abcd" => ""
mynthtail 2 [1, 2, 3, 4] => [3,4]
mynthtail 4 [1, 2, 3, 4] => []

3) myordered

myordered [] => True
myordered [1] => True
myordered [1,2] => True
myordered [1,1] => True
myordered [2,1] => False
myordered "abcdefg" => True
myordered "ba" => False

4) myreplaceall

myreplaceall 3 7 [7,0,7,1,7,2,7] => [3,0,3,1,3,2,3]
myreplaceall 'x' 'a' "" => ""
myreplaceall 'x' 'a' "abacad" => "xbxcxd"

Problem 4: List Comprehension (10 points)

Use list comprehension to write a function 'change' which for any given integer amount computes the optimal (i.e. smallest) set of coins that can make up that amount.

If you couldn't implement it with list comprehension, implement with recursion for 5 points.

Assume the available coins to be 1,2,5,10,20,50,100.

You may use auxiliary functions or you may compute a set of non-optimal answers before selecting the optimal one out of them. These are all fine as long as your core computation (in whichever function it occurs) is done through list comprehension. The list of coins can be hard-coded into your program in any representation you wish.

The only permissible built-in functions are :, head, tail, and null.

change : Int -> [Int]