COMP2209 Assignment Instructions

UNIVERSITY OF Southampton

Module:	*Programming III*			**Examiner:**	*amg@ecs*
Assignment:	*Haskell Programming Exercises*			**Effort:**	45 *hours*
Deadline:	16:00 on 16/11/2017	**Feedback:**	3 *weeks later*	**Weighting:**	30%

Learning Outcomes (LOs)

- Understand the concept of functional programming and be able to write programs in this style.

Instructions

This assignment allows you to demonstrate your understanding of the functional programming techniques covered in the first half of the module, corresponding to Part I of Hutton's Haskell textbook or equivalent. There are 15 functional programming exercises. Each of these exercises will be marked out of 2 using the following scale:

2 / 2: the submitted solution passed all automatically applied tests.

1 / 2: the submitted solution failed one or more of the automatic tests, but is substantially correct.

0 / 2: no solution was provided, or the submitted solution is incorrect.

Note that it is your responsibility to adhere to the instructions specifying the type and behaviour of each function, and your work will not receive full marks if you fail to do so. Likewise your solutions must be pure functions without any side effects. This includes the need to avoid printing any values, as doing so may interfere with the automated tests. You are welcome to define auxiliary functions as needed. It may help you to avoid name clashes if you use a convention such as adding a single quote to the end of the function name, so that for example a function f can use auxiliary functions f', f'' and so on. Your submission may import Data.Char and Data.List, but no other additional Haskell libraries. Be aware that some of these exercises may be more difficult than others. If you get stuck on one problem, you are advised to move on to the next one. Note also that where functions are declared with a type such as Int or [Int] only zero and positive values will be used in the automated tests. In these exercises you are therefore not required to code additional functionality to process negative values in the supplied parameters. (The type Nat is not standard in Haskell, so is not used in these instructions.)

Functional Programming Exercises

Exercise 1

Define a function subtotal, where subtotal :: Num a => [a] -> [a] that adds up all numbers in a given list up to the current position. For example, subtotal [1, 2, 3, 4] = [1, 3, 6, 10].

Exercise 2

Define a function histogram :: Int -> [Int] -> [Int] where histogram n xs which returns a list counting the frequency of values from xs in the range 0..n-1, n..2n-1, 2n..3n-1 and so on until all values in xs have been counted. For example, histogram 5 [1, 2, 10, 4, 7, 12] = [3, 1, 2], and histogram 5 [1, 2, 10, 4, 12] = [3, 0, 2]. Note that the last entry in the result list should be the one corresponding to the range containing the highest value in the list, and that the histogram list entry should be 0 for any range where no values in xs are found.

Exercise 3

Define a function meetsOffer which computes the new UCAS tariff and then decides whether a specific points offer has been met by the given A level grades. For example meetsOffer "A*BC" 150 = False as the tariff value for A*BC is 56+40+32 = 128. In the new tariff system, each A level grade is worth from 16 points (grade E) in increments of 8 to 56 points (grade A*). You may assume that the string of grades is valid and does not contain any spurious characters.

Exercise 4

Define a function sortType xs which returns the value

Ascending if the values in xs are all strictly increasing, or the list is empty or singleton

NonDescending if they increase or stay the same, but are not all the same

Constant if they are all the same,

NonAscending if they decrease or stay the same, but are not all the same, and

Descending if they are all strictly decreasing, or

NotSorted if none of the previous situations apply.

Thus sortType [1, 2, 3] = Ascending, sortType [5, 4, 3, 3, 1] = NonAscending, and sortType [1, 2, 0] = NotSorted. The function must have the type

```
sortType :: Ord a => [a] -> TypeOfSort
```

where

```
data TypeOfSort = Ascending | NonDescending | Constant | NonAscending | Descending | NotSorted
  deriving Show
```

Exercise 5

Define a function rpcalc which takes a string of characters representing a reverse polish calculation and returns the computed value. For example the string "345*-" represents the expression 3 – (4 * 5), and the string "345*-6+7/" represents ((3 – (4 * 5)) + 6) / 7. One way to evaluate such expressions is to process the string from left to right, pushing each digit onto a stack, and executing each operator on the pair of values at the top of the stack and then pushing the result back on to the stack. You can assume that each input number is only a single digit, i.e a single character. All four main arithmetic operators '+' '*' '-' and '/' (integer division, or 'div') must be supported.

Exercise 6

Define a function neighbours k p xs which returns a list of the k nearest (in Euclidean distance) neighbours to p in the list xs of points, represented as two dimensional Cartesian coordinates. This function has the type neighbours :: (Floating a, Ord a) => Int -> (a,a) -> [(a,a)] -> [(a,a)]. If there are fewer than k items in the list xs, then all should be returned as suitable neighbours. If two coordinates in the list are equally close, the earlier value should be returned. Thus neighbours 1 (0.0,0.0) [(1.0,1.0), (1.0,-1.0)] = [(1.0,1.0)]. Your solution may return the selected coordinates in any order.

Exercise 7

Define a function balanced :: SearchTree -> Boolean which checks whether the given tree is a balanced binary search tree. At each node the heights of the left and right-subtrees must differ by at most one. Also the values must be strictly increasing as the tree is traversed using from left to right. The function uses the data type definition

```
data SearchTree = Node SearchTree Int SearchTree | Leaf Int
  deriving show
```

Exercise 8

Use Newton's method to define a function newtonRootSequence :: Double -> [Double]. You can assume the supplied function parameter d is non-negative. The Newton root sequence starts at 1 and each element of the sequence is followed the next approximation where $x_{n+1} = (x_n + d / x_n)$ div 2. Finally, define a function newtonRoot:: Double -> Double -> Double such that newtonRoot d epsilon returns the first value of the sequence which is no more than epsilon different to the previous value in the Newton root sequence for d.

Exercise 9

A sequence of hyper-operators can be defined as follows. The first hyper-operator is +, the second is × and the third is exponentiation. The connection between these operators can be seen in the identity a + a + … + a = a × b whereby adding a to itself b times gives a × b, whereas a × a × … a = a^b so that multiplying a by itself b times gives a raised to the power b. Identify and generalise this pattern and define a function hyperOperator :: Int -> Int -> Int -> Int. For example

hyperOperator 1 4 3 = 4 + 3,

hyperOperator 2 4 3 = 4 × 3,

hyperOperator 3 4 3 = 4 ^ 3, and

hyperOperator 4 4 3 = (4 ^ 4) ^ 4.

(Historic footnote: hyper-operations are closely related to the Ackermann function, which is known to grow faster than any primitive recursive function, and hyper-operations can also easily get very large).

Exercise 10

A character string is encoded for transmission using the following algorithm. Each character is converted to an eight bit string representing its Unicode value. A ninth bit is appended to this list so that the encoded character has even parity, i.e. an even number of 1s. The resulting strings are then concatenated together. For example, the string "ab0" has the encoding [0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,1,0,1,0,0,1,1,0,0,0,0,0]. Define a function encode :: String -> [Int] which implements this encoding. You can assume that each character in the input string has a Unicode encoding less than 256.

Exercise 11

Define a function decode which decodes the bit string described in the previous exercise and returns the original character string. If the given bit string encoding does not represent any character string, then the function should return the empty string "".

Exercise 12

Define a function makeChange m denoms which returns a list with a number of coins of each denomination that total to the given sum of money m. That is to say sum [n * d, (n,d) <- zip (makeChange m denoms) denoms] = m. For example, makeChange 6 [1,5,10] = [1,1,0]. Note that there may be several valid ways of making change, and any which satisfies the identity give here is acceptable. If there is no valid solution, return a list with -1 coins for each denomination, for example [-1,-1,-1] if there are three denominations. You may choose to use dynamic programming to ensure your solution is efficient, though simpler solutions are also acceptable as we will not stress test your solution.

Exercise 13

A generalised natural number representation is given by the number base paired with a sequence of digits in that base. For example, the pair (3, [1, 2, 0, 1]) represents $1\times3^0 + 2\times3^1 + 0\times3^2 + 3^3$. Note that the digits are listed in the reverse of the usual order for simplicity. The (weak) Goodstein successor of a strictly positive number is found by increasing its base by 1 and then subtracting 1 from the resulting number. For example the Goodstein successor of the previous number is (4, [0, 2, 0, 1]). A Goodstein sequence is the sequence of natural numbers using the generalised representation describe above where each after the number after the first is followed by its Goodstein successor. If the successor is zero, then the sequence terminates, otherwise it continues forever. For example the Goodstein sequence starting with (2,[1,1]) is [(2,[1,1]), (3,[0,1]), (4,[3]), (5,[2]), (6,[1]), 7,[]).

The sequence should be empty if the input does not represent a valid positive number in any base. Define a function goodsteinSequence :: (Int, [Int]) -> [(Int, [Int])] so that goodsteinSequence (b,xs) computes the Goodstein sequence starting with generalised natural number (b,xs).

Exercise 14

Define a function isSat to check whether a given Boolean expression is satisfiable and, if so, to return a binding that makes it true. If not, return an empty list. You are advised to start with the tautology checking example given in Section 8.6 of Hutton's textbook on Haskell and the lecture notes. The type should be isSat :: Prop -> [Subst] where the types Prop and Subst are defined as in the lecture notes.

Exercise 15

The Cantor pairing function is defined as follows: pair x y = y + (xysum) * (xysum + 1) 'div' 2 where xysum = x + y. Define a function isCantorPair to test whether a given natural number is a member of the recursive set encoding the (+) function. That is to say, whether the given value is equal to pair (pair x y) (x+y) for some natural numbers x and y. You may need to investigate on-line to find out more about this pairing function and how to invert it.

Submission

Please submit your Haskell solutions as a plain text file called exercises.hs using the ECS hand-in system.

Late submissions will be penalised at 10% per working day. No work can be accepted more than five working days late.
Please also note the University regulations regarding academic integrity and special considerations.