COMP2209 Assignment Instructions

UNIVERSITY OF Southampton

Module:	*Programming III*			**Examiner:**	*amg@ecs*
Assignment:	*Haskell Programming Exercises* (*version* 2)			**Effort:**	15 *to* 45 *hours*
Deadline:	16:00 on 08/11/2017	**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 clear and nearly correct

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

To assist with your implementation, a file with sample test cases is provided, and you should not change the signature of the functions nor the defined types. This imports your Exercises.hs code, and a dummy version of this file is also provided which you can edit to incorporate the code you have developed. You must not, however, change the signatures of these functions nor the defined types, nor add any Import statements. As well as the published test cases an additional test suite will be applied to your code during marking. To prevent anyone from gaining advantage from special case code, this second test suite will only be published after marking has been completed.

It is your responsibility to adhere to the instructions specifying the behaviour of each function, and your work will not receive full marks if you fail to do so. Your code will be tested only on values satisfying the assumptions stated in the description of each exercise, so you can implement any error handling you wish, including none at all. Where the specification allows more than one possible result, any such result will be accepted. When applying the tests it is possible your code will run out of space or time. A solution which fails to complete a test suite for one exercise within 30 seconds on the ECS login server will be deemed to have failed that exercise. Any reasonably efficient solution should take significantly less time than this to terminate on the actual test data that will be supplied.

Note that some of these exercises are more difficult than others. If you get stuck on one problem, therefore, you should move on to the next one. Depending on your proficiency with functional programming, the time required for you to implement and test each solution is expected to be 1 to 3 hours per exercise. If you are spending much longer than this, you are advised to consult the teaching team for advice on coding practices.

Exercises

Exercise 1

Define a function `splitSort :: Ord a => [a] -> [[a]]` that splits the given list into non-empty sub-lists each of which is either in strictly ascending order, in strictly descending order, or contains all equal elements. For example, `splitSort[1,2,3,2,1,1,1] = [[1,2,3],[2,1],[1,1]]`. Note that your implementation should be greedy, meaning that it consumes as many elements as possible to use in the current sub-list before starting a new one.

Exercise 2

Define a function `longestCommonSubList :: Eq a => [[a]] -> [a]` that returns the longest sub-list of each of the finite list of finite lists supplied to the function. For example, `longestCommonSubList [[1,2,3], [0,1,3], [1,3,4]] = [1,3]`. Note that for this exercise, `[1,3]` is defined to be a sub-List of `[1,2,3]`, so that elements of the sub-list do not need to occur in neighbouring positions of the super-list. More precisely `s` is a sub-list of `t` if and only if `s` can result from deleting zero or more members of `t`. Your implementation should satisfy the equation `longestCommonSubList [] = []`. This is valid because any list is a sub-list of all members of an empty list.

Exercise 3

You are asked to implement a function that determines whether or not a University of Southampton undergraduate student has passed their second year and can progress to year 3 according to the standard progression regulations given in section IV of the Calendar. Each module result is provided via a value of type `data ModuleResult = ModuleResult { credit :: Float, mark :: Int} deriving Show`, and the second year outcomes are represented by a list of such values. For example `canProgress [(ModuleResult 40.0 50), (ModuleResult 20.0 50)] = True` whereas `canProgress [(ModuleResult 20.0 50), (ModuleResult 20.0 50), (ModuleResult 20.0 30)] = False` as there are too many credits of failed modules for compensation to apply. You may assume that each module has a pass mark of 40, the qualifying mark in each case is 25, and none of the modules is core. At most 15 credits of failed modules at or above the qualifying mark may be compensated, in which case they are also awarded credit. Note that modules may have credit weightings other than 7.5 ECTS, provided there are at least 60 ECTS credits in total. If fewer credits than this have been taken, then the student has not yet passed and may not progress. For simplicity, assume the function is given the set of marks after any referral or repeat attempts replace earlier results, and with such marks already capped according to the regulations.

Exercise 4

You are asked to implement a function that determines the degree outcome for a University of Southampton undergraduate student who has successfully completed their studies. This could be `First`, `UpperSecond`, `LowerSecond`, or `Third`, depending on their average, based on the regulations in section IV of the Calendar. You can assume there have been either 3 or 4 years of study, and that each year's marks are supplied by a list of module results as in the previous exercise. For example `classify [[ModuleResult 60.0 45], [ModuleResult 60.0 45], [ModuleResult 60.0 45]] = Third` whereas `classify [[ModuleResult 60.0 45], [ModuleResult 60.0 55], [ModuleResult 60.0 65]] = UpperSecond` due to the different weightings of each year. You may assume that each year has been successfully passed. Note that modules may have weights other than 7.5 ECTS credit weightings and different years could, in principle, have different credit totals, each of which is 60 ECTS credits or more. For simplicity, you are not expected to include ordinary degrees in your solution.

Exercise 5

Implement a function `hillClimb` to return an approximation of the local maximum of a given function `f` within the interval from `x0` to `x1`, values which are supplied as its second and third parameters. Your solution should aim to return a value within ε of the true local maximum, where ε is supplied as the fourth parameter. It should use the golden section search method as explained for example on Wikipedia[1] to reduce the search interval efficiently, converging when the width of this interval is less than or equal to $\sqrt{\varepsilon}$.

Exercise 6

Use your solution to the previous exercise to find the nearest root to the given `x` of a polynomial `f(x)`. Note that f^2 is always zero or positive, so that any root of any function `f` is a minimum of f^2, and vice versa. The `nearestRoot` function is supplied a list of coefficients defining the polynomial, so that `nearestRoot [-18.0, 0.0, 2.0] 0.0 5.0 1e-5 ~= 3.0` since the quadratic $2x^2 - 18$ has 3 as its only positive root. Note that the coefficients are supplied in the reverse of the usual order, with the constant value at the head and the coefficient associated with the highest power of `x` as the last value in the list. The second, third and fourth parameters to this function are the same as those for the `hillClimb` function in the previous exercise.

Exercise 7

A zero-address stack-based computer has instructions `Add`, `Multiply`, `Duplicate`, and `Pop` belonging to the data type `Instruction`. Define a function `executeInstructionSequence` which, given a stack and a list of instructions, emulates each instruction and returns the resulting stack. The stack is represented as a list of integers, so the function takes a list of integers and a list of instructions and returns a list of integers. The `Add`, and `Multiply` instructions implement the arithmetic operations with these names, taking the first two values off the stack, performing the operation, and then pushing the result back on to the stack. For example, `executeInstructionSequence [4, 5] [Add] = [9]`. The `Duplicate` instruction takes the value on top of the stack and pushes another copy of this onto the stack. The `Pop` instruction simply removes the value on the top of the stack. Hence `executeInstructionSequence [4, 5, 6, 7] [Pop, Duplicate] = [5, 5, 6, 7]`.

---

[1] https://en.wikipedia.org/wiki/Golden-section_search

Exercise 8

Assuming that the stack has the singleton initial value [x] then the instruction sequence [Duplicate, Multiply] will leave the single value x^2 on the stack. Likewise, [Duplicate, Duplicate, Duplicate, Multiply, Multiply, Multiply] will produce x^4. There is however a more efficient sequence for computing this result, namely [Duplicate, Multiply, Duplicate, Multiply]. Write a function to give an optimal (ie shortest) instruction sequence for raising the initial value x to the supplied parameter n, which you can assume is a positive integer. For example, valid results for this function would include optimalSequence 1 = [], and optimalSequence 2 = [Duplicate, Multiply].

Exercise 9

A busy beaver is an instruction sequence containing only Pop, Multiply and Add instructions that terminates with the highest possible result. This result must be a single value contained in the returned stack, which must have length 1. For example, the instruction sequence [Pop] gives the maximum result possible when executed on the input [0,1], as the returned stack is [1]; the sequence [Add] also gives the same result. The findBusyBeavers function returns the list of all busy beavers for the given input stack s. Hence a valid result for this function would be findBusyBeavers [0,1] = [[Pop],[Add]], or indeed the same list in a different order.

Exercise 10

In this exercise, a rectangle is represented by a pair of integer valued coordinates identifying its top-right and bottom-left corners. If the top-right corner is below or to the left of the second corner, the rectangle is empty. These rectangles all have horizontal and vertical edges. You are given a list of such rectangles representing a black and white image constructed using these rectangles. Each rectangle's corners and edges are considered to be part of the image, as well as all points with integer valued coordinates in each rectangle's interior. If the rectangles overlap, there may be a shorter list that represents the same image. For example, [Rectangle (0,0) (2,2), Rectangle (0,0) (1,1)] can be simplified to [Rectangle (0,0) (2,2)] since the second rectangle is wholly contained within the first one. More complex scenarios are possible. Write a function simplifyRectangleList that gives a minimal sequence which represents the same image as the originally supplied rectangle list. You do not need to implement an optimal solution, as this requires sophisticated data structures, but your solution is expected to have polynomial time complexity rather than exponential.

Exercise 11

You are asked to generate a list of rectangles giving an image of an ellipse. The ellipse is defined by the floating point input parameters xCentre yCentre a b and has a boundary logically consisting of those real-valued points (x,y) satisfying the equation $(x - xCentre)^2 / a^2 + (y - yCentre)^2 / b^2 = 1$. In practice, not all such points can be represented computationally. The image generated by these rectangles should contain the (integer) coordinates of all points on or within the given ellipse. It is also expected that the list by the drawEllipse function is a minimal one in the sense defined in the previous exercise.

Exercise 12

This exercises concerns a steganography technique used to hide a secret message inside a text file. The hiding technique is to replace a letter O with a digit 0, and likewise the letter I with the digit 1. For simplicity our secret message is assumed to use only the letters a, b, c and d, each of which is encoded by the bit string 00, 01, 10, and 11, respectively. For example, to send the message "bad", the bit string 010011 needs to be hidden in the text file. Suppose the text file contains HI HOW ARE YOU DOING? I AM DOING FINE, OK! IS IT TIME TO GO? This is then changed to HI H0W ARE YOU DO1NG? I AM D0ING FINE, 0K! 1S 1T TIME TO GO? Careful reading of the text file extracts the bit string 010011, which is then decoded to give "bad", the secret message. You are asked to write a function extractMessage which scans the text provided, extracts the bit string, then decodes this to give the secret message. You may assume that the secret message has been correctly inserted, and that the original text file included no digits prior to hiding the message.

Exercise 13

Define a function differentStream, that returns a stream of binary digits which is not present in the stream of streams supplied to the function. For example, differentStream [[1..], [2..], [3..], ..] = [0..] would be a valid result. You may assume that the actual parameter passed to this function is an infinite stream of infinite streams of integers. Note that the existence of this function shows that, for example, $\mathbb{N} \rightarrow \mathbb{N}$ is uncountable.

Exercise 14

The square shell pairing function was defined by Rosenberg and Strong in 1972, and analysed by Szudzik (2017)[2]. This pairing function is illustrated by the table below. It takes two natural numbers as arguments and produces a single natural number as its result (thereby proving that $\mathbb{N}\times\mathbb{N}$ is the same size as $\mathbb{N}$, and hence countable). For example, the value shown at position (3, 2) is 13 which is in the fourth and final square shell illustrated here, so that `Pair 3 2 = 13`.

…	…	…	…	…	…
3	9	10	11	12	…
2	4	5	6	13	…
1	1	2	7	14	…
0	0	3	8	15	…
	0	1	2	3	…

You are asked to define a function `unPairAndApply` the supplied natural number `n` to a pair `(x,y)` by inverting the pairing function described above, and then returns the result of applying the function `f` provided as its second parameter to the two values resulting from unpairing `n`. Hence `unPairAndApply 13 (–) = 1` since `3 – 2 = 1`. Note that the function `f` is supplied in curried form, so that it should be applied using `f x y` rather than `f(x,y)`.

Exercise 15

The pairing function from the previous exercise can also be used to represent a binary tree as a single natural number. For example, given the recursively defined type `data Tree = Empty | Node Int Tree Tree deriving Show` the `Empty` tree can be represented by the number `0`, and the tree `Node n t u` by the number `Pair n (Pair t1 u1)` where the number `t1` represents the tree `t` and the number `u1` represents the tree `u`. For this exercise, you are asked to implement a function `isShellTreeSum` which tests whether its supplied argument `n` is the result of pairing `x` and `y` where `x` is the number representing a tree `t` and `y` is the result of summing the values at each node of `t`. For example, the integer `3` represents the tree `Node 1 Empty (Node 0 Empty Empty)` which sums to `1`. Moreover the entry at position (3, 1) in the table above is 14, and hence `isShellTreeSum 14` is `true`.

Submission

Please submit your Haskell solutions as a plain text file called `Exercises.hs` using the ECS hand-in system. This will be marked automatically as described above. Solutions which do not pass all the tests for a given exercise will be reviewed and given a mark of 0 or 1 for that exercise, using the criteria stated above.

The code you submit must have been written by you, so you must not copy code from other students. Where you adapt code you have taken from the Internet, you must cite your source or sources. To discourage unethical behaviour, your code will be analysed using plagiarism detection software.

Note also that supplying your code to anyone else will also be considered a breach of academic integrity as doing so will be interpreted as encouraging other students to plagiarise. This restriction therefore prohibits you from emailing your code to fellow students or sharing it on forums such as Stack Overflow. It is moreover clearly unacceptable to use such a forum to ask other people to debug code you have written but are unable to get working yourself. Instead you are expected to ask the teaching team for coding and debugging advice during the scheduled lab sessions. You may also describe your code and the errors you receive to fellow students and ask for their advice, provided you do not actually show them your code.

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.

[2] Szudzik, M. P. (2017). The Rosenberg-Strong Pairing Function.