Final Sample Problems

Problem 0(a) The *Fusion Law* states that under certain conditions we may conclude that

```
f . foldr g a = foldr h b
```

These conditions are

(i)

(ii)

(iii)

Problem 0(b) Express (double . sum) as (foldr h b) by identifying appropriate choices for h and b.

Problem 1 Define a function

```
thatD :: [[Int]] -> [Int]
```

which takes a square matrix and returns in a list the items along its diagonal. Do not use the indexing operator `!!`. Examples:

```
thatD [[2]] = [2]
thatD [[2,1],[3,8]] = [2,8]
thatD [[2,1,3],[4,1,3],[8,1,1]] = [2,1,1]
```

(You need not check for squareness of the input.)

Problem 2 Define a function

```
subD :: [[Int]] -> [Int]
```

which takes a square matrix and returns in a list the items along its *subdiagonal*. Do not use the indexing operator !! . Examples:

```
subD [[1]] = []
subD [[1,2],[2,1]] = [2]
subD [[1,2,3],[4,5,6],[7,8,9]] = [4,8]
```

(You need not check for squareness of the input.)

Problem 3 Define a function

```
lowerT :: [[Int]] -> [Int]
```

which takes a square matrix and returns in a list the items on or below the diagonal. Do not use the indexing operator !!. Examples:

```
lowerT [[1]] = [1]
lowerT [[1,2],[2,1]] = [1,2,1]
lowerT [[1,2,3],[4,5,6],[7,8,9]] = [1,4,5,7,8,9]
```

(You need not check for squareness of the input. You may return the items in any order you wish.)

Problem 4 Below is an incomplete sequence of steps in a proof by induction of the identity

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Fill in the missing justifications.

Case x:xs

```
((x:xs) ++ ys) ++ zs

= {                                  }

(x:(xs ++ ys)) ++ zs

= {                                  }

x:((xs ++ ys) ++ zs)

= {                                  }

x:(xs ++ (ys ++ zs))

= {                                  }

(x:xs) ++ (ys ++ zs)
```

Problem 5 Below is an incomplete sequence of steps in a proof by induction of the identity

```
reverse (ys ++ [x]) = x:reverse ys
```

Fill in the missing expressions.

Case y:ys

reverse ((y:ys) ++ [x])

= {++.2}

= {reverse.2}

= {induction}

= { ++.2 }

= { reverse.2 }

Problem 6(a) Suppose that `f = until ((==1) . length) (reverse . tail)`. Give the type signature for `f`.

Problem 6(b) With the same definition for f as above, apply f to the list

```
[1..444]
```

Problem 6(c) Suppose I want to remove some parentheses from the definition given in part (a). Consider these compositions:

```
((==1) . length)
(reverse . tail)
```

Can I remove the outer parentheses without changing the meaning or syntactical correctness of the definition for f?

(i) No. All parentheses are needed.

(ii) Parentheses can be safely removed from `((==1) . length)` only.

(iii) Parentheses can be safely removed from `(reverse . tail)` only.

(iv) Parentheses can be safely removed from both compositions.

Problem 7 Define a function

```
threeDifferent :: Int -> Int -> Int -> Bool
```

so that the value of threeDifferent m n p is True exactly when all three of the numbers m, n and p are different from each other.

Problem 8 Using a list comprehension, define a function

```
ordPairs :: Int -> [(Int,Int)]
```

such that ordPairs n is a list of all integer pairs (x,y) satisfying $0 < x < y < n$. Do not assume that n is a natural number.

Consider the function

```
foldr :: (a -> a -> a) -> [a] -> a
foldr f [x]    = x
foldr f (x:xs) = f x (foldr f  xs)
```

Given this definition...

Problem 9(a) Evaluate foldr (||) [False,True,False]

Problem 9(b) Evaluate foldr (++) ["Don't ", "Panic", "", "!"]

Problem 9(c) Evaluate foldr (*) [1 .. 6]

Problem 10 Suppose we declare a new type

```
data Name = Short String | Fullname String String
```

Write a function

```
title :: Name -> String
```

that takes a full name and returns it prepended with the word "Esteemed", or, given a short name as input, simply returns that name. For example:

```
title (Fullname "Irwin" "Cohen") = "Esteemed Irwin Cohen"
title (Short "Irwin")  = "Irwin"
```

Problem 11 Consider the expressions below

(i) [0,1)

(ii) if 1==0 then 2==1

(iii) [[ ] , [[ ]] , [[[ ]]]]

Which of these is valid? Circle your response.

(I) Only (iii)

(II) Both (iii) and (ii), but not (i)

(III) All three

(IV) None of them

Problem 12 *From the Edinburgh course:*

Suppose the playing cards in a standard deck are represented by characters in the following way: `'2'` through `'9'` plus `'0'` (zero) stand for the number cards, with `'0'` representing the 10, while the `'A'`, `'K'`, `'Q'` and `'J'` stand for the face cards, i.e. the ace, king, queen and jack, respectively. Let's call these the 'card characters'. The other characters, including the lowercase letters `'a'`, `'k'`, `'q'`, and `'j'`, and the digit `'1'`, are not used to represent cards.

(a) Write a function `f :: String -> Bool` to test whether all card characters in a string represent face cards. For example:

```
f "ABCDE" = True      f "none here" = True     f "4 Aces" = False
f "01234" = False     f "" = True              f "1 Ace" = True
```

Your function can use *basic functions*, *list comprehension* and *library functions*, but not recursion. Credit may be given for indicating how you have tested your function.

(b) Write a function `g :: String -> Bool` that behaves like `f`, this time using *basic functions* and *recursion*, but not library functions or list comprehension. Credit may be given for indicating how you have tested your function.

(c) Write a function `h :: String -> Bool` that behaves like `f` using one or more of the following higher-order functions:

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr :: (a -> b -> b) -> b -> [a] -> b
```

You can also use *basic functions*, but not other library functions, recursion or list comprehension. Credit may be given for indicating how you have tested your function.

Problem 13 *From the Edinburgh course:*

(a) Write a function `t :: [a] -> [a]` that duplicates every other item in a list. The result should contain the first item once, the second twice, the third once, the fourth twice, and so on. For example,

```
t "abcdefg" = "abbcddeffg"
t [1,2,3,4] = [1,2,2,3,4,4]
t ""        = ""
```

Your definition may use *basic functions*, *list comprehension*, and *library functions*, but not recursion. Credit may be given for indicating how you have tested your function.

(b) Write a second function `u :: [a] -> [a]` that behaves like `t`, this time using *basic functions* and *recursion*, but not list comprehension or library functions. Credit may be given for indicating how you have tested your function.

Problem 14 *From the Edinburgh course:*

(a) Write a function `f :: [Int] -> [Int]` that produces a list of distances between consecutive numbers in a list, in those cases where the first number is less than the second number. For example:

```
f [4,2,5,6,1,8] = [3,1,7]
f [] = []
f [3] = []
f [3,3,1,-3] = []
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. Credit may be given for indicating how you have tested your function.

(b) Write a second function `g :: [Int] -> [Int]` that behaves like `f`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions*. Credit may be given for indicating how you have tested your function.

Problem 15 *From the Edinburgh course:*

(a) Write a function `f :: [Int] -> [String]` that compares consecutive numbers in a list and, when they are different, indicates whether the first is less than (`"<"`) or greater than (`">"`) the second. For example:

```
f [4,2,5,6,1,8] = [">","<","<",">","<"]
f [] = []
f [3] = []
f [3,3,1,-3] = [">",">"]
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. Credit may be given for indicating how you have tested your function.

(b) Write a second function `g :: [Int] -> [String]` that behaves like `f`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions*. Credit may be given for indicating how you have tested your function.

Problem 16 *From Prof Royer:*

The standard Haskell function

```
all :: (a -> Bool) -> [a] -> Bool
```

is such that `(all tst xs)` returns `True` if and only if, when the function `tst` is applied to each element of `xs`, all of the results are `True`.
E.g.: `(all even [2,4,6])` $\rightsquigarrow$ `True`, `(all even [2,3,6])` $\rightsquigarrow$ `False`, and `(all even [])` $\rightsquigarrow$ `True`.

(a) Write, `allRec`, a recursive version of `all`.

(b) Write, `allFold`, a version of `all` using `foldr`.

Problem 17 *From Prof Royer:*

BACKGROUND. A list xs is a *sublist* of the list ys if xs can be obtained by removing some number (possibly zero) of the elements from ys.

For example, `[1,5,3,1]` is a sublist of `[1,2,5,1,3,1]` (= `[`**`1`**`,2,`**`5`**`,1,`**`3`**`,`**`1`**`]`). In contrast, `[1,3,5,1]` is *not* a sublist of `[1,2,5,1,3,1]`, as it cannot be obtained simply by removing some elements of `[1,2,5,1,3,1]`.

YOU PROBLEM. Write a Haskell function

```
sublist :: (Eq a) => [a] -> [a] -> Bool
```

such that `(sublist xs ys)` determines whether or not xs is a sublist of ys. (***Hint:*** The stopping cases involve empty lists.) For example, your function should have the following behavior:

```
Main> sublist [1,5,3,1] [1,2,5,1,3,1]
True
Main> sublist [1,3,5,1] [1,2,5,1,3,1]
False
Main> sublist [1,2,3,4] [1,2,5,1,3,1]
False
```

Problem 18 *On Induction:*

Prove by induction on xs that ++ is *associative*, i.e.:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

for all finite lists xs and all lists ys and zs.

Problem 19 *On Time Complexity:*

Derive the time complexity for the function implemented below. You might as well give an induction proof so that you are not guessing at it.

```
subseqs' (x:xs) = xss ++ map (x:) xss
                  where xss = subseqs' xs
```

Problem 20 *On `IO`*

Write an interactive Haskell program using the IO monad which prompts the user for a name, takes user input from the keyboard, and then greets the user with the given name. For example, your program prints “Who are you?", then I type ’Martin’, then your program prints “Oh, hello Martin!".

Problem 21 *On Limits & Approximations*

Recall from Bird's chapter on infinite lists the following definition:

```
approx :: Integer -> [a] -> [a]
approx n []       | n>0 = []
approx n (x:xs)   | n>0 = x:approx (n-1) xs
```

Prove that approx does the thing it is meant to do:

$$\lim_{n\to\infty} \text{approx n xs} = \text{xs}$$

for *all lists finite, partial or infinite.*

Problem 22 *On Parsing (Exercise G from Bird)*

Design a parser for recognising Haskell floating-point numbers. Bear in mind that `.314` is not a legitimate number (no digits before the decimal point) and that `3 . 14` is not legitimate either (because no spaces are allowed before or after the decimal point).