CS_221 Functional Programming I

(Attempt 2 questions out of 3)

Preliminary remark: By a *function* we mean a *Haskell function*, and by the *definition of a function* the *signature* of that function (that is, the statement of the type of the function) followed by its *defining equations*.

Question 1

(a) **(i)** What are the types of the following values?

(i-1) `('a','b')`

(i-2) `['a','b']`

(i-3) `[tail]`

(ii) What are the types of the following functions?

(ii-1) `pair x y   =  (x,y)`

(ii-2) `list2 x y  =  [x,y]`

(ii-3) `diag f x   =  f x x`

(iii) What are the values of the following expressions?

(iii-1) `length (filter even [1..5])`

(iii-2) `map length [[x..3] | x <- [1..3]]`

(iii-3) `(\g -> \x -> g (g x)) (* 3) 2`

[9 marks]

(b) Define a function `evens` that takes as inputs an integer `n` and a function `f :: Int -> Int`, and computes the list of all positive integers `x` below `n` such that `f x` is even.

[8 marks]

(c) Consider the following function:

```
inc :: [Int] -> Bool
inc []     = True
inc [x]    = True
inc (x:xs) = x < head xs && inc xs
```

(i) Describe informally what this function does.

(ii) Generalise the signature of this function using a suitable type constraint. Explain informally what the generalised signature means.

[8 marks]

Question 2

(a) **(i)** Briefly describe Haskell's evaluation strategy.

(ii) Name an advantage and a disadvantage of this strategy.

[7 marks]

(b) In Haskell's Prelude file the function `uncurry` is defined as follows:

```
uncurry        :: (a -> b -> c) -> ((a,b) -> c)
uncurry f p    = f (fst p) (snd p)
```

Consider the following variant:

```
uncurry1            :: (a -> b -> c) -> ((a,b) -> c)
uncurry1 f (x,y)    = f x y
```

Explain why `uncurry` and `uncurry1` are *not* equivalent under Haskell's evaluation strategy. Give an example

```
f x y = ...
p = ...
```

such that the expressions `uncurry f p` and `uncurry1 f p` behave differently when evaluated.

[8 marks]

(c) **(i)** Consider the following definition:

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (ys ++ zs)
```

Prove

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

by list induction on `xs`.

(ii) Consider the following definitions:

```
data Tree = Leaf Int | Branch Tree Tree

flatten :: Tree -> [Int]
flatten (Leaf x)       = [x]
flatten (Branch t1 t2) = (flatten t1) ++ (flatten t2)

flatten1 :: Tree -> [Int] -> [Int]
flatten1 (Leaf x) xs       = x : xs
flatten1 (Branch t1 t2) xs = flatten1 t1 (flatten1 t2 xs)
```

Prove

```
flatten1 t xs = flatten t ++ xs
```

by tree induction on `t`.

[10 marks]

Question 3

(a) Define a function

```
partition :: Int -> [a] -> [[a]]
```

such that for a positive integer `n` and a list `xs`, `partition n xs` partitions `xs` into parts of length `n` where the last part might be shorter than `n`. For example, `partition 3 [1,2,3,4,5,6,7]` should yield `[[1,2,3],[4,5,6],[7]]`.

Hint: Define `partition n xs` by recursion on `xs` using the library functions

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

[7 marks]

(b) Suppose that a polymorphic abstract data type of *finite sets* is to be implemented by *repetition-free lists*:

```
type Set a = [a]
```

(i) Define, as part of this implementation, a function

```
intersect :: Eq a => Set a -> Set a -> Set a
```

that computes the intersection of two sets.

(ii) Suppose we restrict the type parameter `a` to types for which an ordering, `<`, is defined, that is, we require the type `a` to be a member of the type class `Ord`.

Give a more efficient implementation of the function `intersect` for sets represented by repetition-free *ordered lists*.

(iii) Estimate the run time complexities of the functions you defined in (i) and (ii).

[10 marks]

(c) **(i)** Briefly describe how functions with side effects can be programmed in Haskell. Give an example of a function with side effect.

(ii) Briefly explain Haskell's `do`-notation. What is it syntactic sugar for?

[8 marks]