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* we mean the *defining equations* of that function preceded preceded by its *signature*.

Question 1.

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

(i) `map even [1..5]`

(ii) `filter even [1..5]`

(iii) `[x^2 | x <- [1..5], even x]`

(iv) `(\f -> \x -> (f (f x))) (^2) 3`

[8 marks]

(b) Define a function that computes, for a positive integer n, the list of its divisors, that is the list of all positive integers d such that d divides n. For any unsuitable input an error shall be raised.

[9 marks]

(c) Consider the following functions

```
zero :: (Int,Int) -> Int
zero (x,y) = 0

zero1 :: (Int,Int) -> Int
zero1 p = 0
```

Explain why the functions `zero` and `zero1` are *not* equivalent.

How is this fact connected with Haskell's evaluation strategy?

Would the results be different if a different evaluation strategy were used?

Hint: Consider the reductions for the expressions `zero silly` and `zero1 silly` where

```
silly :: (Int,Int)
silly = silly
```

[8 marks]

Question 2.

(a) 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 `subset` that tests inclusion of sets. The function `subset` should be defined for every type `a` for which an equality test, `==`, is defined, that is, the type `a` must be member of the type class `Eq`. Therefore the signature of `subset` is

```
subset :: Eq a => Set a -> Set a -> Bool
```

(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 `subset` for sets represented by repetition-free *ordered* lists.

(iii) Estimate the run time complexities of the functions you defined in part 2 (a) (i) and 2 (a) (ii) in terms of the number of elements of sets assuming that an equality test and a comparison between two elements each cost one time unit.

[10 marks]

(b) Write a program `findCode :: IO ()` that finds, for every character `c`, input by the user as a string of length 1, its number, that is, the value of `fromEnum c :: Int`. The program shall terminate when the user inputs the empty string, that is, presses Return without entering anything.

Hints: In order to get the user's input use `getLine :: IO String`, for printing the result use the functions `show :: Show a => a -> String` and `putStrLn :: String -> IO ()`.

[7 marks]

(c) Consider the following definitions:

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

comp :: (b -> c) -> (a -> b) -> (a -> c)
comp f g x = f (g x)
```

Prove

```
map f (map g xs) = map (comp f g) xs
```

by induction on lists.

[8 marks]

Question 3.

(a) Suppose a point in the 2-dimensional plane is represented by a pair of floating point numbers, and a polygon is represented by a list of points:

```
type Pt = (Float,Float)
type Polygon = [Pt]
```

(i) Define a function that rotates a polygon by an angle θ around the origin where the angle is given in radians (represented by floating point numbers).

(ii) Define a function that scales a polygon `p` by a given factor `z :: Float`, that is, every coordinate of a point in `p` is multiplied by `z`.

[8 marks]

(b) (i) Define a polymorphic data type `Tree a` of binary trees with leaves labeled by elements of some unspecified type `a`.

(ii) Define a polymorphic higher-order function `mapTree` that takes as arguments a function `f :: a -> b` and a tree `t :: Tree a` and computes the tree which is built like `t`, but where each label `x` is replaced by `f x`.

[9 marks]

(c) Assume we represent a real polynomial by its list of coefficients:

```
type Polynomial = [Float]
```

Hence a list $[c_0, c_1, c_2 \ldots, c_n]$ represents the polynomial

$$c_0 + c_1 * X^1 + c_2 * X^2 + \ldots + c_n * X^n.$$

Define an *evaluation function* for polynomials, `evalPol :: Polynomial -> Float -> Float`, that computes, for every polynomial $[c_0, c_1, c_2 \ldots, c_n]$ and floating point number x, the number

$$c_0 + c_1 * x^1 + c_2 * x^2 + \ldots + c_n * x^n$$

Hints: You may use the predefined Haskell function `zipWith` which has the property that

$$\texttt{zipWith}\ g\ [c_0, \ldots, c_n]\ [x_0, \ldots, x_n] = [g\ c_0\ b_0,\ \ldots,\ g\ c_n\ b_n]$$

Alternatively you may use the observation that

$$c_0 + c_1 * x^1 + c_2 * x^2 + \ldots + c_n * x^n = c_0 + x * (c_1 + c_2 * x^1 + \ldots + c_n * x^{n-1})$$

to give a simple recursive definition of the function `evalPol` (Horner's rule).

[8 marks]