Answers Exam Functional Programming – Nov. 4th 2019

1. **Types** (5× 2=10 points)
   (a) Is the following expression type correct? If your answer is `YES`, then give the type of the expression.

   ```
   'a':'b':[]:[]
   ```

   ```
   NO
   ```

   (b) Is the following expression type correct? If your answer is `YES`, then give the type of the expression.

   ```
   ('a':'b':[]):[]
   ```

   ```
   YES, [String]  or [[Char]]
   ```

   (c) What is the most general type of the following function `f`?

   ```
   f = (\x -> \y -> \z -> [x y, x (x z)])
   ```

   ```
   f :: (a -> a) -> a -> a -> [a]
   ```

   (d) What is the most general type of the following function `g`?

   ```
   g = .not
   ```

   ```
   g :: (Bool -> a) -> Bool -> a
   ```

   (d) What is the most general type of the following function `h`?

   ```
   h = not.
   ```

   ```
   h :: (a -> Bool) -> a -> Bool
   ```

2. **Programming in Haskell** (10 points)
   A *well formed string of parentheses* is defined by the following recursive rules:

   - The empty string is well formed.
   - If s is a well formed string, then (s) is a well formed string.
   - If s and t are well formed strings, then their concatenation st is a well formed string.

   For example, `"((()))"` and `"()()()"` are well formed strings, while `"(()"`, `")(()"` and `")("` are not. Write a Haskell function `isWFS :: String -> Bool` such that `isWFS str` returns `True` if the string `str` is well formed and `False` otherwise.

   ```
   isWFS :: String -> Bool
   isWFS str = parse 0 0 str
     where
       parse lpar rpar []       = lpar == rpar
       parse lpar rpar ('(':xs) = parse (lpar+1) rpar xs
       parse lpar rpar (')':xs) = (rpar < lpar) && parse lpar (rpar+1) xs
       parse _ _ _              = False
   ```

3. **Higher order functions** (3+3+4=10 points)

- Without using recursion or a list comprehension, write a function `selectiveMap` which takes three arguments. Also, give the type of the function `selectiveMap`. The first argument of the function is a predicate `p`, the second some function `f`, and the third a list `xs`. The function `selectiveMap` returns a list that is just like `xs`, but in which every element `x` that satisfies `p` is replaced by `f` applied to `x`.
  For example, the call `selectiveMap even (*2) [1,2,3,4,5,6]` should return `[1,4,3,8,5,12]`.

```
selectiveMap :: (a ->Bool) -> (a -> a) -> [a] -> [a]
selectiveMap p f xs = map select xs
  where select x = if p x then f x else x
```

- Without using recursion or a list comprehension, write a function `thresholdPairs` which takes two arguments. The first is an `Integer n`, and the second is a list `xs` of `Integer` pairs. The output should be the list of pairs `(a,b)`, in the same order as in the list `xs`, for which the sum of `a` and `b` is greater than `n`.
  For example, the function call `thresholdPairs 3 [(1,2),(2,2),(3,5),(0,3),(0,4)]` should return `[(2,2),(3,5),(0,4)]`.

```
thresholdPairs :: Integer -> [(Integer,Integer)] -> [(Integer,Integer)]
thresholdPairs n pairs = filter (\(x,y) -> (x+y) > n) pairs
```

- Implement the standard function `map` using the standard function `foldr`.

```
map f xs = foldr (\e ys -> f e:ys) [] xs
```

4. **List comprehensions** (3+3+4=10 points)

- Use a list comprehension to implement the function `partition` which takes two arguments. The first is some element `x`, and the second a list `xs`. The function should return a pair of lists of which the first is the list of all elements of `xs` that are less than or equal to `x`, while the second is the list of all elements of `xs` that are greater than `x`. Also, give the most general type of the function `partition`.

```
partition :: Ord a => a -> [a] -> ([a],[a])
partition x xs = ([y | y <- xs, y<=x], [y | y <- xs, y > x])
```

- Use an efficient list comprehension to implement the function `tripletSum` (including its type) that takes a positive `Integer n`, and returns the lexicographically ordered list of all triples `(a,b,c)` such that `n` equals `a+b+c` and `1<=a<=b<=c`. For example, `tripletSum 6` should return `[(1,1,4),(1,2,3),(2,2,2)]`.

```
tripletSum :: Integer -> [(Integer,Integer,Integer)]
tripletSum n = [(a,b,n-a-b) | a <- [1..n `div` 3], b <- [a..(n-a) `div` 2]]
```

- The function `adjacentTriples` takes a list `xs` and outputs the list of all triples of adjacent elements in the list `xs`. Give its type and an implementation using a list comprehension. For example, `adjacentTriples "curry"` should return `[('c','u','r'),('u','r','r'),('r','r','y')]`.

```
adjacentTriples :: [a] -> [(a, a, a)]
adjacentTriples xs = [(a,b,c) | (a,(b,c)) <-
                                   zip xs (zip (drop 1 xs) (drop 2 xs))]
```

5. **infinite lists** (3+3+4=10 points)

- Give a recursive implementation of the function `iterate` (including its type) that takes two arguments. The first is a function `f` and the second some value `x`. The call `iterate f x` returns an infinite list of repeated applications of `f` to `x`. So, `iterate f x=[x, f x, f(f x), f(f(f x)), f(f(f(f x))),....]`.
  For example, `take 10 (iterate (*2) 1)` yields `[1,2,4,8,16,32,64,128,256,512]`.

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

- Give a definition of the infnite list `tribonacci` which is the ordered list of all tribonacci numbers. Recall that the tribonacci numbers are defined as: $T(n) = n$ for $n < 3$ and $T(n) = T(n-1) + T(n-2) + T(n-3)$ for $n \geq 3$.
  So, `take 10 tribonacci` should return `[0,1,2,3,6,11,20,37,68,125]`. Your implementation must make (useful) use of the function `zipWith`, so `map T [0..]` is not accepted as a valid answer.

```
tribonacci = 0:1:2:zip3 tribonacci (drop 1 tribonacci) (drop 2 tribonacci)
  where zip3 xs ys zs = zipWith (+) xs (zipWith (+) ys zs)
```

- Give a definition of the infnite list `palindromes` which is a list of lists of palindromic bit strings. The nth list contains all lexicographically sorted palindromes of length n (starting with $n = 0$). For example, `take 4 palindromes` produces `[[""],["0","1"],["00","11"],["000","010","101","111"]]`.

```
palindromes = [[]]:["0","1"]:
                [[a:(xs++[a]) | a<-['0','1'], xs <- xss] | xss <- palindromes]
```

6. (15 points) The type `Polynomial` is an Abstract Data Type (ADT) for real valued polynomials.
Implement a `module Polynomial` such that the implementation of the type `Polynomial` is hidden to the user.

The following operations need to be implemented:

- `makePolynomial coeffs` converts the coeffcients in the list `coeffs` into a `Polynomial`. For example, `makePolynomial [2.0,0.0,0.5]` should produce the `Polynomial` representation of $2x^2 + 0.5$.
- `eval pol x` returns the evaluation of the polynomial `pol` at `x`. For example, $2x^2 + 0.5$ at $x = 1.0$ can be computed using `eval (makePolynomial [2.0,0.0,0.5]) 1.0`.
- `add lhs rhs`: returns the polynomial that is the addition of `lhs` and `rhs`. For example, $(2x^2 + 0.5) + (x - 1)$ can be constructed using `add (makePolynomial [2.0,0.0,0.5]) (makePolynomial [1.0,-1.0])`.
- `scale a pol`: multiplies the polynomial `pol` by the scalar `a`. For example, $5(2x^2 + 0.5)$ can be constructed using `scale 5.0 (makePolynomial [2.0,0.0,0.5])`.

```
module Polynomial (Polynomial, makePolynomial, eval, add, scale) where

data Polynomial = P [Double]

makePolynomial :: [Double] -> Polynomial
makePolynomial coeff = P (reverse coeff)

eval :: Polynomial -> Double -> Double
eval (P coeff) x = sum(zipWith (*) coeff [x^e | e <- [0..]])

add :: Polynomial -> Polynomial -> Polynomial
add (P lhs) (P rhs) = P(psum lhs rhs)
  where
    psum (x:xs) (y:ys) = (x+y):psum xs ys
    psum xs ys = xs++ys

scale :: Double -> Polynomial -> Polynomial
scale s (P coeff) = P [s*c | c <- coeff]
```

7. **Proof of equality** (10 points) Consider the following Haskell function.

```
f 0 = 0
f 1 = 1
f n = 5*(f (n-1)) - 6*(f (n-2))
```

Prove that `f n` = $3^n - 2^n$ for all non-negative integers `n`.

```
The property is easily proved using natural induction on n.
Base cases (n=0, and n=1):
  f 0 = 0 = 1 - 1 = 3^0 - 2^0
  f 1 = 1 = 3 - 2 = 3^1 - 2^1
Induction step: Assume that the property holds for n.
    f (n+1)
  = {definition f}
```

```
    5*(f n) - 6*(f (n-1))
  = {use hypothesis twice}
    5*(3^n-2^n) - 6*(3^(n-1)-2^(n-1))
  = {arithmetic: note that 6=3*2}
    5*(3^n-2^n) - (2*3^n-3*2^n)
  = {arithmetic}
    3*3^n - 2*2^n
  = {arithmetic}
    3^(n+1) - 2^(n+1)    QED.
```

8. **Proof on trees** (15 points) Given is the data type `Tree` and the functions `inorder`, and `flatten`:

```
data Tree a = Empty | Node a (Tree a) (Tree a)

inorder :: Tree a -> [a]
inorder Empty = []
inorder (Node x l r) = inorder l ++ [x] ++ inorder r

flatten :: Tree a -> [a] -> [a]
flatten Empty ys = ys
flatten (Node x l r) ys = flatten l (x:flatten r ys)
```

Prove for all finite trees `t`: `inorder t = flatten t []`
[Note: If you need one or more lemmas to complete the proof, then prove these lemmas separately.]

```
The property follows immediately from the following lemma (with ys=[]):
   flatten t ys = inorder t ++ ys

We prove this property using structural induction on Trees.
Base case (t=Empty):
   flatten Empty ys
 = {def. Flatten}
   ys
 = {def. ++}
   []++ys
 = {def. inorder}
   indorder Empty ++ ys

Inductive step (t=Node x l r): assume that the property holds for l and r.
   flatten (Node x l r) ys
 = {def. flatten}
   flatten l (x:flatten r ys)
 = {ind. hypothesis for r}
   flatten l (x:(inorder r ++ ys))
 = {def. ++}
   flatten l ((x:inorder r) ++ ys)
 = {ind. hypothesis for l}
   inorder l ++ ((x:inorder r) ++ ys)
 = {lemma: for any xs we have x:xs = [x]++xs}
   inorder l ++ (([x] ++ inorder r) ++ ys)
 = {lemma (twice): associativity of ++}
   (inorder l ++ [x] ++ inorder r) ++ ys
 = {def. inorder}
   inorder (Node x l r) ++ ys

The first lemma is proved without induction: [x]++xs=(x:[])++xs=x:([]++sx)=x:xs

The 2nd lemma is a standard one: xs++(ys++zs)=(xs++ys)++zs with the consequence that
parentheses can be plaved arbitrarily. This lemma is proved by induction.
Base: []++(ys++zs)=ys++zs=([]++ys)++zs
Inductive case: (x:xs)++(ys++zs)=x:(xs++(ys++zs))
                =x:((xs++ys)++zs)=(x:(xs++ys))++zs=((x:xs)++ys)++zs
QED.
```