**Q 1 [30%]**

**a) allEqual xs [10%]**
**is every item in the list xs equal to every other such item? (return True if xs has only 0 or 1 items)**

```
allEqual :: Eq a => [a] -> Bool
allEqual []  = True
allEqual [_] = True
allEqual ( x1:x2:xs ) = x1 == x2 && allEqual( x2:xs )
```

**b) allDifferent xs [10%]**
**is every item in the list xs different from every other such item? (return true if xs has only 0 or 1 items)**

```
allDifferent :: Eq a => [a] -> Bool
allDifferent [] = True
allDifferent [_] = True
allDifferent ( x1:x2:xs ) = x1 /= x2 && allDifferent( x2:xs )
```

**c) countMax xs [10%]**
**the number of times that its maximum item occurs in the non-empty list xs; this function should make only one pass over the list**

```
countMax :: [Int] -> Int
countMax [] = 0
countMax (x:xs) = countMax' x 1 xs

countMax' :: Ord a => a -> Int -> [a] -> Int
countMax' _ count [] = count
countMax' max count (x:xs) = if max == x then
                                countMax' max (count + 1) xs
                             else if max > x then
                                countMax' max count xs
                             else
                                countMax' x 1 xs
```

**Q 2 [35%]**

**a) Consider the standard Haskell function id, defined by: [5%]**

```
id = x
```

**State the (most general) type of id.**

(a)
```
id :: a -> a
```

(b)
```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

(c)
```
compose :: [a -> a] -> a -> a
```

(d)
```
compose [] x = x
compose (f:fs) x = f ( compose fs x )
```

(e)
```
compose fs x = foldr (\f -> \input -> f input ) x fs
```

**Q 3 [35%]**

**a) Consider the standard Haskell function zipWith, where: [11%]**

**zipWith f [x1, x2, ...] [y1, y2, ...] => [f x1 y1, f x2 y2, ...]**

**The length of the output list is the smaller of the lengths of the two input lists; the output list is infinite if both input lists are. Give a recursive definition for ZipWith, including its type.**

```
zipWith :: ( a -> a -> a ) -> [a] -> [a] -> [a]
zipWith f xs ys = if null xs || null ys then
                                        []
                  else
                      f( head xs ) ( head ys ) :
                      zipWith f( tail xs ) ( tail ys )
```

**b) Consider the Haskell definition: [12%]**

**mystery = 1 : zipWith (*) mystery [1..]**

**Write down the value of the expression:**

**take 5 mystery**

**Then suggest a more meaningful name for mystery and state its type.**

take 5 mystery => [1, 1, 2, 6, 24]
A more meaningful name would be 'factorials'.
factorials :: [Int]

**c) Recall that a Pythagorean Triple consists of a 3-tuple of positive integers x, y, z with x < y < z and x^2, y^2, z^2. [12%]**

**[ (x,y,z) | x <- [1..], y <- [1..], z <- [1..], x < y, y < z, x*x, + y*y == z*z ]**

**Explain why this attempt is incorrect, and write a corrected version.**

The order is wrong: We are given x < y, however we define x before we define y.
The limits are wrong for x,y and z.

Correct Answer from James

Candidate items for a list expressed by a list comprehension are generated first and then checked by the boolean guards (e.g. x*x + y*y == z*z) that filter out unwanted items.
In the definition above, x, y and z are specified by their generator expressions (e.g. x <- [1..]), and they each start at 1.
The last generator (for z) must be exhausted before the previous generator (for y) can move onto the next value, so the unfiltered list of (x, y, z) tuples created by these generators is: [(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 1, 4),..]

Values for z are taken from the infinite list [1..], so an infinite amount of work must be done before the value of y can be incremented to 2; similarly, y must go through an infinite number of values before x can be incremented.
The first Pythagorean triple is the tuple (3, 4, 5); however, this item will never be generated because neither x or y can move on from 1.

Correct version

```
pyTrips = [ (x, y, z) | z <- [5..], y <- [4..(z-1)], x <- [3..(y-1)], (x * x) + (y * y) == (z * z) ]
```

Here he didn't say about x & y not having common factors like he did in the assignment version of the question. In case that was a mistake...

```
pyTrips = [ (x, y, z) | z <- [5..], y <- [4..(z-1)], x <- [3..(y-1)], (x * x) + (y * y) == (z * z), gcd x y == 1 ]
```

**Q1. Explain the concept of a curried function and describe, with the help of an example, one of its advantages.**

Currying is the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed.

**Q2. Explain the concept of lazy evaluation, and give an example of a Haskell function whose observed behaviour would differ if Haskell instead used eager evaluation.**

Objects are only computed when you need them, when you're writing in Haskell you can feel free to generate and use infinitely long lists, and the compiler will only do the work to compute the ones you use. e.g fibs

Operations are not performed until their results are actually needed.

Call-by-Need: When a function is applied to an argument, the argument remains unevaluated until its value is needed within the function.

Lazy Definitions: When a definition is processed, its <EXPRESSION> is not actually evaluated, and thus can contain occurrences of <NAME>s which have yet to be defined.

**Q3. Explain the difference between imperative and functional programming.**

Functional Programming

- Works without the concepts of variables and assignment
- No notion of time

Imperative Programming

- Works with the concepts of variables and assignment
- Program progresses over time -> timing is crucial
- Sequence of instructions

A **higher-order function** is a function that takes other functions as arguments or returns a function as result .e.g. zipWith.