IN3043 Functional Programming
Solutions to Exercises 7

1. An example evaluation:

```
foo [1,2,2,1,1,1,3] ⇝ 1 : bar 1 [2,2,1,1,1,3]
                    ⇝ 1 : 2 : bar 2 [2,1,1,1,3]
                    ⇝ 1 : 2 : bar 2 [1,1,1,3]
                    ⇝ 1 : 2 : 1 : bar 1 [1,1,3]
                    ⇝ 1 : 2 : 1 : bar 1 [1,3]
                    ⇝ 1 : 2 : 1 : bar 1 [3]
                    ⇝ 1 : 2 : 1 : 3 : bar 3 []
                    ⇝ 1 : 2 : 1 : 3 : []
```

   The function **foo** returns the original list with multiple adjacent duplicate values collapsed to a single entry.

2. (a) There are three cases:

```
removeFirstDigit [] = []
removeFirstDigit (x:xs)
  | isDigit x    = xs
  | otherwise    = x : removeFirstDigit xs
```

   (b) We generalize the previous answer from the specific predicate **isDigit** to a variable **p**:

```
removeFirst p [] = []
removeFirst p (x:xs)
  | p x          = xs
  | otherwise    = x : removeFirst p xs
```

3. (a) We could do this with four cases:

```
addLists :: Num a => [a] -> [a] -> [a]
addLists [] [] = []
addLists [] (y:ys) = y:addLists [] ys
addLists (x:xs) [] = x:addLists xs []
addLists (x:xs) (y:ys) = (x+y):addLists xs ys
```

   Alternatively, once one of the lists is empty, we can just return the other one:

```
addLists :: Num a => [a] -> [a] -> [a]
addLists [] ys = ys
addLists xs [] = xs
addLists (x:xs) (y:ys) = (x+y):addLists xs ys
```

(b) We parameterize the previous definition over a function **f**:

```
longZip :: (a -> a -> a) -> [a] -> [a] -> [a]
longZip f [] ys = ys
longZip f xs [] = xs
longZip f (x:xs) (y:ys) = f x y:longZip f xs ys
```

4. (a) If either list is empty, the other is the answer. If both are non-empty, the head of the result is the smaller of the head of the lists, and the other head is retained for further merging:

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x < y     = x:merge xs (y:ys)
  | otherwise = y:merge (x:xs) ys
```

The test could have been `x <= y` without changing the meaning.

(b) You can do this by defining

```
odds :: [a] -> [a]
odds [] = []
odds [x] = [x]
odds (x1:x2:xs) = x1:odds xs

evens :: [a] -> [a]
evens [] = []
evens [x] = []
evens (x1:x2:xs) = x2:evens xs
```

Note that **`[x]`** means a list of exactly one element, which we're calling **`x`**. These three patterns cover empty lists, lists with one element and lists with at least two elements respectively.

But the neatest way to do this is to make these functions depend on each other:

```
odds :: [a] -> [a]
odds [] = []
odds (x:xs) = x:evens xs
```

```
evens :: [a] -> [a]
evens [] = []
evens (x:xs) = odds xs
```

(c) Empty lists and lists of one element are easy to sort:

```
mergeSort [] = []
mergeSort [x] = [x]
```

Any other list must be non-empty, so we can use the suggested method:

```
mergeSort xs =
    merge (mergeSort (odds xs)) (mergeSort (evens xs))
```

Note that the empty list case is essential; without it, the recursion never stops.