# G6021: Comparative Programming

Exercise Sheet 6

## 1 Higher-Order functions on lists

1. Write a function `sumAll` that returns the sum of all the elements of a list of numbers.

   **Answer:**

   ```
   sumAll [] = 0
   sumAll (h:t) = h+sumAll t
   ```

2. Write a function `multAll` that returns the product of all the elements of a list of numbers.

   **Answer:**

   ```
   multAll [] = 1
   multAll (h:t) = h*multAll t
   ```



3. The pattern of recursion should be the same in the previous two questions: the only differences are the name of the function, the function applied to each of the elements, and the starting value. Haskell provides a function to capture this kind of recursion: `foldr f b l`, where the three parameters are the function to be applied, the starting value and the list.



   Examples of this functions in use include:



   ```
   foldr (*) 1 [1,2,3,4]
   foldr (+) 0 [1,2,3,4]
   ```

   Test these functions, and check that they give the same answers as your functions above.

4. Write your own version of the `foldr` function (call this function `fold`).

   **Answer:**

   ```
   fold f b [] = b
   fold f b (h:t) = f h (fold f b t)
   ```

5. Using `fold`, write the following:

   - A function `len` to compute the length of a list.

     **Answer:**

     ```
     len = fold (\x -> \y -> 1+y) 0
     ```

   - A function `maxElem` to compute the maximum element of a list.

     **Answer:**

     ```
     maxElem = fold (\x -> \y -> if x>y then x else y) 0
     ```

   - A function `flatten` to convert a list of lists to a list. Example: `flatten [[1,2,3],[4,5,6]]` should give `[1,2,3,4,5,6]`.

     **Answer:**

```
flatten = fold (++) []
```

6. What is the type of the function `iter` below, and what do you think it does? What does the function `f` compute?

```
iter p f x = if (p x) then x else iter p f (f x)

f n = snd(iter (\(x,y) -> x>n) (\(x,y) -> (x+1,x*y)) (1,1))
```

**Answer:**

```
iter :: (t -> Bool) -> (t -> t) -> t -> t
```

This function `iter` accumulates repeated applications of the function: `f(...(f x))`, until the given testing function stops the iteration.
The function `f` is computes the factorial of the given argument. It is also interesting to trace what is happening to the tuple (1,1):

```
(1,1) -> (2,1) -> (3,2) -> (4,6) -> (5,24) -> (6,120) ->  ....
```

## 2 Accumulating parameters

Test out the three versions of factorial from the notes: `fact` and `factcps` and `factacc`.



**Answer:**

```
fact n = if n==0 then 1 else n*fact(n-1)
fact 4
```



```
factcps n k = if n==0 then k 1
              else factcps (n-1) (\r -> k (n*r))
factcps 4 (\x -> x)
```



```
factacc n acc = if n==0 then acc
                else factacc (n-1) (n*acc)
factacc 4 1
```

1. Write a version of the `len` function using an accumulating parameter (so that the function is tail recursive):

```
len [] = 0
len (h:t) = 1+len t
```

**Answer:**

```
lenacc [] acc = acc
lenacc (h:t) acc = lenacc t (acc+1)
```

2. Write a version of the `rev` function using an accumulating parameter:

```
rev [] = []
rev (h:t) = (rev t) ++ [h]
```

**Answer:**

```
revacc [] acc = acc
revacc (h:t) acc = revacc t (h:acc)

revacc [1,2,3] []
```

## 3 Data types

1. Give a datatype, called `IntOrBool`, that can represent either an `Int` or a `Boolean`. Show how you can use this datatype to represent a list of mixed elements (Integers and Booleans).

   **Answer:**

   ```
   data IntOrBool = I Int | B Bool deriving (Show)
   [I 3,B True]
   ```

2. Suggest a way to represent the $\lambda$-calculus as a data type in Haskell.

   **Answer:**

   ```
   data Lam = Var [Char] | Abs [Char] Lam | App Lam Lam deriving (Show)
   ```