University of Canberra
Faculty of Information Sciences and Engineering

Computer Engineering 1 — 1997
Part 1 : Introductory Programming

Practical Class 5 — Week 6

Aim

The aim of this class is to give you more practice defining functions on lists, especially using recursion and list patterns.

Exercise 1 — Correcting a definition

The following function definition is faulty. It does *not* do what its description claims. Correct the definition so that the function fulfils its stated purpose. Try to work out what is wrong *before* trying it in Hugs.

```
delete y xs returns the list xs with the
first occurrence of item y removed.

> delete :: Eq a => a -> [a] -> [a]

> delete y [] = []
> delete y (x:xs)
>   | x==y       = xs
>   | otherwise  = delete y xs
```

Exercise 2 — Variations on selection

A reasonably common list operation is selecting those elements of a list that all have some property. The following function is one such example. Observe that there are two alternative recursive expressions in this definition, only one of which is used in any one recursive cycle. The selection is based on the relationship between the value of `y` and the value of `x`, which is (currently) the head of the list `(x:xs)`.

```
select y xs extracts from the list xs, all the items whose value is
less than y.

> select :: Ord a => a -> [a] -> [a]

> select y [] = []
> select y (x:xs)
>   | x < y      = x : select y xs
>   | otherwise  = select y xs
```

Construct the following function definitions. They are all variations on the theme of the `select` function above:

1. Define a function to return the *number of times* a given value `y` occurs in a list `xs`.

2. Define a function to *delete all instances* of a given value `y` from a list `xs`.

3. Define a function to *replace each occurrence* of the value `y` in the list `xs` with the value `z`.

   Using this function, write another function to replace each space character in a string with a newline character `'\n'`, so that the string will be printed one word per line.

4. Define a function to *replace only the first occurrence* of the value `y` in the list `xs` with the value `z`.

Exercise 3 — Searching a sorted list
There is a standard Haskell function `elem`, which determines whether a given value (`y`, say) occurs in a list of elements (`xs`, say) of the same type. If `y` does not actually appear in the list `xs`, the function still has to search right through the list to ascertain that fact.

If the list of elements is already sorted into ascending order, then searching can cease when the element is found, or when we know that the value of each remaining element is greater than the value of the element being searched for. Write a function `elemSort y xs` which assumes that the list `xs` is already sorted and which implements this more efficient searching technique.

Exercise 4 — Removing duplicates from a sorted list
In the Haskell library module `List.hs`, there is a function `nub` which removes multiple occurrences of values from a list.

As in the previous exercise, this is defined to work for lists in general. If the list is sorted, we can take advantage of the fact that multiple occurrences of values will be adjacent to each other in the list. Define a function `remDups xs` which removes duplicates from the list, but assumes that the list is sorted.

Exercise 5 — Detecting duplicates in a sorted list
Write a function that determines if there are duplicate elements in a list. Your function should take advantage of the fact that the list is sorted, so that duplicates are adjacent if they exist.

Submission
You have now completed the main exercises. Your solutions should all be in the same script. Make sure to show your solutions to your tutor to have them assessed.

Extension — Detecting duplicates in an unsorted list
Write a function that determines if there are duplicate elements in a list without assuming the list is sorted. There are many different ways to achieve this. You may find the standard function `elem` useful. For another approach, you may find that `nub` can be put to use.