

Week 5 Practical Class Exercises

Recursive Functions on Lists

Objectives

The aim of these exercises is to give you more practice defining your own functions on lists, especially using recursion and list patterns. You may not have time to complete all of these exercises during your lab class. If not, you should finish them in your own time before next week's class.

Exercise 1 (Correcting a definition)

The following function definition is defective. 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* experimenting with it in GHCi. (Hint: what is the value of `deleteFirst 4 [1..10]` ?)

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

deleteFirst :: Eq a => a -> [a] -> [a]
deleteFirst y [] = []
deleteFirst y (x:xs)
  | x == y      = xs
  | otherwise   = deleteFirst y xs
```

There is a script `DelWrong.hs` containing this definition on the COMP1100 web site.

Exercise 2 (Variations on selection)

A 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`.

```
-- 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 (Adding two polynomials)

A polynomial in a single variable can be represented rather simply by a list of its coefficients (which will be `Floats`). For example:

<code>[1, 7, 5, 2]</code>	represents	$2x^3 + 5x^2 + 7x + 1$
<code>[42, 2, 1]</code>	represents	$x^2 + 2x + 42$
<code>[-3, 0, 0, 0, 1]</code>	represents	$x^4 - 3$
<code>[0, -2, 0, 4]</code>	represents	$4x^3 - 2x$

Notice how the list index for each element corresponds to the exponent of the term. Define a type synonym `Poly` for this representation.

Two polynomials can be summed by adding the coefficients of corresponding terms. For example, the sum of $2x^3 + x^2 + 1$ and $3x^4 + 4x^2 - 7$ is $3x^4 + 2x^3 + 5x^2 - 6$.

Define a Haskell function `sumPoly :: Poly -> Poly -> Poly` that sums two polynomials that are represented as above. Take care with the case of polynomials of different degrees. For example:

```
*Polynomial> sumPoly [1, 7, 5, 2] [42, 2, 1]
[43.0,9.0,6.0,2.0]
*Polynomial> sumPoly [-3, 0, 0, 0, 1] [1, 7, 5, 2]
[-2.0,7.0,5.0,2.0,1.0]
```

This function takes two list arguments so think about the list patterns that will be useful in the definition of `sumPoly`.

Make sure that your script includes appropriate comments.

Exercise 4 (Evaluating a polynomial)

We also want a Haskell function which, given a polynomial and a value for `x`, will calculate the value of the polynomial for that value of `x`. For example:

```
*Polynomial> evalPoly 3 [1, 7, 5, 2]
121.0
*Polynomial> evalPoly (-2) [0, -2, 0, 4]
-28.0
```