CA320 Computability and Complexity

Lab 4: Defining More Functions on Lists

Aim

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

1. Deleting Values From a List

Define a recursive function `delFirst :: Eq a => a -> [a] -> [a]` that deletes the first occurrence of a given value from a list. For example:

```
> delFirst 2 [1,2,3,2,1]
[1,3,2,1]
```

To check your answer is correct (and to register your progress), you should save your program to a file named `delFirst.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

Define a recursive function `delAll :: Eq a => a -> [a] -> [a]` that deletes all occurrences of a given value from a list. For example:

```
> delAll 2 [1,2,3,2,1]
[1,3,1]
```

To check your answer is correct (and to register your progress), you should save your program to a file named `delAll.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

2. Number of Occurrences of a Value in a List

Define a recursive function `num :: Eq a => a -> [a] -> Int` that gives the number of occurrences of a given value in a list. For example:

```
> num 2 [1,2,3,2,1]
2
```

To check your answer is correct (and to register your progress), you should save your program to a file named `num.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

If the list of elements is already sorted into ascending order, then counting occurrences can stop when we reach a value that is greater than the value being counted. Define a recursive function `numSorted :: Ord a => a -> [a] -> Int` that takes advantage of this property to give the number of occurrences of a given value in a sorted list. For example:

```
> numSorted 2 [1,1,2,2,3]
2
```

To check your answer is correct (and to register your progress), you should save your program to a file named `numSorted.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

3. Replacing Values in a List

Define a recursive function `repFirst :: Eq a => a -> a -> [a] -> [a]` that replaces the first occurrence of one value with another in a list. For example:

```
> repFirst 2 4 [1,2,3,2,1]
[1,4,3,2,1]
```

To check your answer is correct (and to register your progress), you should save your program to a file named `repFirst.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

Define a recursive function `repAll :: Eq a => a -> a -> [a] -> [a]` that replaces all occurrences of one value with another in a list. For example:

```
> repAll 2 4 [1,2,3,2,1]
[1,4,3,4,1]
```

To check your answer is correct (and to register your progress), you should save your program to a file named `repAll.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

4. List Member

There is a standard Haskell function `elem :: Eq a => a -> [a] -> Bool`, that determines whether a given value occurs in a list of elements of the same type. For example:

```
> elem 2 [5,4,3,2,1]
True
```

If the value does not actually appear in the list, 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 stop when the element is found, or when we know that the value of the current element is greater than the value of the element being searched for. Define a recursive function `elemSorted :: Ord a => a -> [a] -> Bool` that assumes the list is already sorted and implements this more efficient searching technique. For example:

```
> elemSorted 2 [1,2,3,4,5]
True
```

To check your answer is correct (and to register your progress), you should save your program to a file named `elemSorted.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

5. Removing Duplicates From a List

In the Haskell library module `Data.List`, there is a function `nub :: Eq a => [a] -> [a]` that removes multiple occurrences of values from a list. For example:

```
> nub [1,2,3,2,1]
[1,2,3]
```

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 recursive function `nubSorted :: Eq a => [a] -> [a]` that removes duplicates from the list, but assumes that the list is sorted. For example:

```
> nubSorted [1,1,2,2,3]
[1,2,3]
```

To check your answer is correct (and to register your progress), you should save your program to a file named `nubSorted.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

6. Detecting Duplicates in a List

Define a recursive function `dupSorted :: Eq a => [a] -> Bool` that determines if there are duplicate elements in a sorted list. Your function should take advantage of the fact that the list is sorted, so that duplicates are adjacent if they exist. For example:

```
> dupSorted [1,1,2,2,3]
True
```

To check your answer is correct (and to register your progress), you should save your program to a file named `dupSorted.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein

Define a recursive function `dup :: Eq a => [a] -> Bool` that determines if there are duplicate elements in a list, without assuming that the list is sorted. For example:

```
> dup [1,2,3,2,1]
True
```

To check your answer is correct (and to register your progress), you should save your program to a file named `dup.hs`, and drag and drop this file to the upload link on the following page:
https://ca320.computing.dcu.ie/einstein