MATHFUN
Discrete Mathematics and Functional Programming

Worksheet 5: List Patterns and Recursion

Introduction

This worksheet aims to extend your skills in writing list-processing functions, by using list patterns and recursion on lists. Begin by copying the Week5.hs file from the unit website to your file-space. This file defines a module Week5 containing most of the functions from the lecture. Load this script into GHCi, and experiment by evaluating some expressions. Add the solutions to the programming exercises to the script.

List patterns

1. Write a function:

   ```
   headPlusOne :: [Int] -> Int
   ```

   which returns the first element plus one for a non-empty list, and 0 for an empty list. (e.g. `headPlusOne [5,7,2,4] = 6`.)

2. Write a polymorphic function:

   ```
   duplicateHead :: [a] -> [a]
   ```

   which adds an extra copy of the first element at the beginning of the list (or returns `[]` for an empty list). (For example, `duplicateHead [5,7] = [5,5,7]`.)

3. Write a polymorphic function:

   ```
   rotate :: [a] -> [a]
   ```

   which swaps the first two elements of a list (or leaves the list unchanged if it contains fewer than two elements). (For example, `rotate [5,7,2,4] = [7,5,2,4]`.)

Recursion over lists

4. Write a recursive polymorphic function:

   ```
   listLength :: [a] -> Int
   ```

   which returns the length of a list (i.e. a re-implementation of the Prelude's `length` function).

5. Write a recursive function:

   ```
   multAll :: [Int] -> Int
   ```

   which returns the product of all the integers in the list (or 1 for an empty list). (For example, `multAll [5,2,4] = 40`.)

6. Write a recursive function:

   ```
   andAll :: [Bool] -> Bool
   ```

   which returns the conjunction (and) of all the elements of a list (e.g. `andAll [True, True] = True`, and `andAll [True, False] = False`.) The function should return True for the empty list.

7. Write a recursive function::

   ```
   countElems :: Int -> [Int] -> Int
   ```

   which counts the number of times a given value appears in a list. (For example, `countElems 3 [5, 3, 8, 3, 9] = 2`). (Hint: This function may require two recursive cases.)

8. Write a recursive function::

   ```
   removeAll :: Int -> [Int] -> [Int]
   ```

   that removes all copies of a given value from a list of integers. (e.g, `removeAll 3 [5, 3, 8, 3, 9] = [5, 8, 9]`).

9. Recall the `StudentMark` type synonym from last week. Write a recursive function:

   ```
   listMarks :: String -> [StudentMark] -> [Int]
   ```

   which gives a list of the marks for a particular student; for example:

   ```
   listMarks "Joe" [("Joe", 45), ("Sam", 70), ("Joe", 52)] = [45,52]
   ```

10. Write a recursive function:

    ```
    prefix :: [Int] -> [Int] -> Bool
    ```

    which decides if the first list is a prefix of the second list (e.g. `[1,4]` is a prefix of `[1,4,9,2]`, and `[]` is a prefix of any list.)

11. [Harder] Using your `prefix` function, write a recursive function:

    ```
    subSequence :: [Int] -> [Int] -> Bool
    ```

    which decides if the first list is contained in the second (e.g. `[1,4,9]` is a subsequence of `[8,1,4,9,2,1]`, and `[]` is subsequence of any list.)