M21274 – 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. If you are confused by a function/operator, look it up in Hoogle.

First copy `Week5.hs` to your usual folder. This file defines the `Week5` module that includes the functions of the lecture notes. Load it into GHCi and experiment by evaluating some expressions. Then work your way through the worked examples before attempting the programming exercises. We do not give the signature of functions in this worksheet, so this is something you need to determine yourself from each question.

Worked example 1

Using recursion, write a function `countSpaces` that counts the number of space characters in a string.

Test	Output
`countSpaces "May the Force be with you."`	5
`countSpaces "Hello_world"`	0
`countSpaces ""`	0

Solution: The function takes a `String` and returns a whole number (`Int`). So add the following at the end of `Week5.hs`:

```
countSpaces :: String -> Int
```

We are going to approach this task with a recursive function that uses pattern matching. Let us begin with the base case (when the string is empty):

```
countSpaces :: String -> Int
countSpaces "" = 0
```

You could also use an empty list (i.e., "`countSpaces []`") as strings are lists of characters. For the general case, we need to match the input string using the prepend operator ":". Read `(x:xs)` as "*a string with first character* `x` *and remainder* `xs`".

```
countSpaces :: String -> Int
countSpaces "" = 0
countSpaces (x : xs)
  | x == ' ' = 1 + countSpaces xs
  | otherwise = countSpaces xs
```

Notice how we add one to the count when the first character (`x`) is equal to a space. This definition should satisfy all test cases the table.

Worked example 2

Using recursion, write a function `mergeLists` that takes two sorted lists of integers and returns a sorted list that contains all the elements of the two lists in order (see the table

below).

Test	Output
`mergeLists [1,3,5] [2,4]`	`[1,2,3,4,5]`
`mergeLists [10,20,30] [1,2,3]`	`[1,2,3,10,20,30]`

Solution: First the base cases, when either (or both) of the argument lists are empty:

```
mergeLists :: [Int] -> [Int] -> [Int]
mergeLists [] ys = ys
mergeLists xs [] = xs
```

Now, consider for example the test case `mergeLists [1,3,5] [2,4]`. Since `1 <= 2`, we need to **prepend** `1` as the head of our output list. For the tail of our output list, we ignore the `1` and call `mergeLists` again (the rest of the output is defined from `mergeLists [3,5] [2,4]`). Similar logic applies for the other case (i.e. if the head of the first list is bigger than the head of the second list.) Eventually one of the arguments becomes empty and instead of making a recursive call, a base case is used. Here is our definition for the general case:

```
mergeLists :: [Int] -> [Int] -> [Int]
mergeLists [] ys = ys
mergeLists xs [] = xs
mergeLists (x:xs) (y:ys)
  | x <= y = x : mergeLists xs (y:ys)
  | otherwise = y : mergeLists (x:xs) ys
```

After trying the test cases from the table, find out what happens when our arguments have common elements by evaluating the following expression:

```
mergeLists [1,2,3] [1,2,3]
```

Programming exercises

List patterns

Write all your definitions inside `Week5.hs` and test them using test data from the tables (and your own). **Do not use anything that we have not covered** so far (e.g., don't use `foldr`, `map` or `filter`).

1. Write a function `headPlusOne` that, given a list of integers, takes the first element, adds one to it and returns the result. The function should return `-1` if the input list is empty.

Test	Output
`headPlusOne [5,4,3]`	`6`
`headPlusOne [-5]`	`-4`
`headPlusOne []`	`-1`

2. Write a polymorphic function called `duplicateHead` that adds an extra copy of the first element at the beginning of a given list (or returns the empty list unchanged):

Test	Output
`duplicateHead [5,7]`	`[5,5,7]`
`duplicateHead "Hello world!"`	`"HHello world!"`
`duplicateHead []`	`[]`

3. Write a polymorphic function `rotate` that swaps the first two elements of a list (or leaves the list unchanged if it contains fewer than two elements).

Test	Output
`rotate [5,7,2,4]`	`[7,5,2,4]`
`rotate "Hello world!"`	`"eHllo world!"`
`rotate [12]`	`[12]`

Recursion over lists

4. Write a recursive polymorphic function `listLength` that returns the length of a list. This is a re-implementation of the Prelude's `length` function, so don't use this in your solution.

Test	Output
`listLength [2.10, 3.20, 4.30]`	`3`
`listLength "Hello world!"`	`12`

   Hint: What is the base case? What should `listLength` return given `[]`?

5. Write a recursive function `multAll` that returns the product of all integers in a list.

Test	Output
`multAll [2, 3]`	`6`
`multAll [2, 3, 4]`	`24`
`multAll []`	`1`

6. Use the `&&` operator to write a recursive function `andAll` which returns the conjunction (and) of all the elements of a list. This is a reimplementation of the Prelude's `and` function so do not use this function in your solution.

Test	Output
`andAll [True, True]`	`True`
`andAll [True, True, False]`	`False`
`andAll []`	`True`

   Note: You do not need guards or if expressions for this function. Your definition should be very similar to `multAll` or `sum`.

7. Write a recursive `orAll` function that returns the disjunction of all elements in the given list (uses the `||` operator). Hint: Start by thinking about how the following expression should be evaluated:

```
orAll []
```

8. Write a function `countIntegers` that counts how many times a given integer appears in a list of integers. This function will be similar in structure to worked example 1.

Test	Output
`countIntegers 3 [5, 3, 8, 3, 9]`	`2`
`countIntegers 7 [5, 3, 8, 3, 9]`	`0`

9. Write a recursive function `removeAll` that removes all instances of a given number from a list of integers.

Test	Output
`removeAll 3 [5, 3, 8, 3, 9]`	`[5, 8, 9]`
`removeAll 7 [5, 3, 8, 3, 9]`	`[5, 3, 8, 3, 9]`

10. Using `removeAll`, write a `removeAllButFirst` function.

Test	Output
`removeAllButFirst 3 [5, 3, 8, 3, 9]`	`[5, 3, 8, 9]`
`removeAllButFirst 3 [3, 4, 5]`	`[3, 4, 5]`

11. Recall the `StudentMark` type synonym from last week. Add the following lines to `Week5.hs` if your copy does not already contain them:

```
type StudentMark = (String, Int)

testData :: [StudentMark]
testData =
  [ ("John", 53),
    ("Sam", 16),
    ("Kate", 85),
    ("Jill", 65),
    ("Bill", 37),
    ("Amy", 22),
    ("Jack", 41),
    ("Sue", 71)
  ]
```

Now, write a function that gives a list of the marks for a particular student.

Test	Output
`listMarks "Joe" [("Joe", 45), ("Sam", 70), ("Joe", 52)]`	`[45,52]`
`listMarks "Amy" testData`	`[22]`

12. Write a recursive function `sorted` which decides if the elements of a list are sorted (the output needs to be of type `Bool`).

Test	Output
`sorted [1,4,4,5]`	`True`
`sorted [1,4,5,4]`	`False`

13. Write a recursive function `prefix` which decides if the first list of integers is a prefix of the second list.

Test	Output
`prefix [1, 4] [1, 4, 9, 2]`	`True`
`prefix [1, 4] [2, 1, 4, 9, 2]`	`False`

Hint: An empty list is the prefix of any list, and no list (except the empty list) is the prefix of an empty list.

14. [Harder] Using your `prefix` function, write a recursive function `subSequence` that decides if the first list is contained in the second.

Test	Output
`subSequence [1, 4, 8] [1, 4, 9, 2]`	`False`
`subSequence [1, 4, 9] [2, 1, 4, 9, 2]`	`True`