Program Design and Data Structures, 20.0 c

Course code: 1DL201, Report code: DL201, 67%, DAG, NML,
week: 44 - 02 Semester: Autumn 2017
week: 03 - 11 Semester: Spring 2018

LAB ASSIGNMENT 5

This information is not available in English. Now showing the Swedish version.

After this lab you should be able to ...

- Write functions that use pattern matching and recursion to process lists.
- Use ranges and list comprehension.
- Measure and compare evaluation time for Haskell expressions.

Instructions

1. Read all slides from lectures 8-9 (Lists) and lecture 10 (Sorting).
2. All problems in this assignment *must* be solved using recursion (except where stated otherwise). Should you know of some "clever" way of solving the problem without recursion, it will not be accepted! Always think about how you can solve a problem by using simpler (smaller) cases of the problem, since that is the basic idea behind recursion.
3. Remember to write function specifications for all (non-anonymous) functions that you write. Do this *before* you write your Haskell code. Also remember to follow the other parts of our coding convention (for identifiers and indentation), and to give a variant for each recursive function.
4. Remember to *test* your functions. Where we have given examples in the assignments, make sure that your programs can run them correctly.
5. The answers to all questions should be *written down* - preferably in a file!

The Task

PROBLEM 1

Write a function myInit :: [a] -> [a] that returns all but the last element of a (non-empty) list. For example,

- myInit [1,2,3] == [1,2]
- myInit ["first","last"] == ["first"]

Use recursion. (The Haskell Prelude already provides a similar function init. Do *not* use this function to solve this problem. Moreoever, do not use take, drop, reverse, or other auxiliary functions.)

Find a variant that proves termination of myInit.

Hints: What is the base case? What is the recursion scheme? Remember that you can combine an element x and an existing list xs with : to obtain a new list x:xs.

PROBLEM 2

Write a function fromDecimals :: [Integer] -> Integer that converts a list of decimals (integers from 0 to 9) into the corresponding integer value. For example,

- fromDecimals [4,2] == 42
- fromDecimals [1,3,3,7] == 1337
- fromDecimals [] == 0

Use an auxiliary (recursive) function that employs an accumulator.

Find a variant that proves termination of your auxiliary function.

Hints: What is the base case? What is the recursion scheme? What should be the initial value of the accumulator?

PROBLEM 3

1. Write a function squareOfEven1 :: [Integer] -> [Integer] that returns the squares of all even numbers in a given list. For example,

- `squareOfEven1` [0,1,2,3,4] == [0,4,16]

Use recursion.

2. Write a function `squareOfEven2` :: [Integer] -> [Integer] that returns the squares of all even numbers in a given list. For example,

- `squareOfEven2` [0,1,2,3,4] == [0,4,16]

Do *not* use recursion. Use list comprehension instead.

PROBLEM 4

Type :set +s at the GHCi prompt. This will display some stats after evaluating each expression, including the elapsed time and number of bytes allocated.

1. Evaluate the following expressions (where insertionSort is defined below), and take note of the respective time required:

- length (insertionSort [1..1000])
- length (insertionSort [1..2000])
- length (insertionSort [1..3000])
- length (insertionSort [1..4000])

- length (insertionSort [1..5000])
- length (insertionSort [1..6000])
- length (insertionSort [1..7000])
- length (insertionSort [1..8000])
- length (insertionSort [1..9000])
- length (insertionSort [1..10000])

2. Evaluate the following expressions (where mergeSort is defined below), and take note of the respective time required:

- length (mergeSort [1..10000])
- length (mergeSort [1..100000])
- length (mergeSort [1..1000000])

Visualize your measurements in two diagrams (one for insertion sort, one for merge sort) that show the list length on the x axis and the evaluation time on the y axis.

Can you explain your findings?

The following code defines merge sort and insertion sort (taken from Lecture 10)

```
split :: [a] -> ([a],[a])
split xs =
  let
    l = length xs `div` 2
  in
   (take l xs, drop l xs)

merge :: [Integer] -> [Integer] -> [Integer]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | y < x = y : merge (x:xs) ys
  | otherwise = x : merge xs (y:ys)

mergeSort :: [Integer] -> [Integer]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs =
  let
    (xs1,xs2) = split xs
  in
   merge (mergeSort xs1) (mergeSort xs2)

insert k [] = [k]
```

```
insert k (x:xs) =
  if k < x then
    k : x : xs
  else
    x:(insert k xs)

insertionSortAux sorted [] = sorted
insertionSortAux sorted (x:xs) =
  insertionSortAux (insert x sorted) xs

insertionSort xs =
  insertionSortAux [] xs
```

WHEN YOU ARE DONE

When you are done with all problems (the explorations below are optional), raise your hand or approach an assistant to have your solution graded.

If you pass this lab at least 30 minutes early and other groups are still working on it, we ask you to **help one other group**. Do not simply share your solution with them, but try to understand the (partial) solutions that they have developed so far (which may be different from yours) and the difficulties that they have. Assist them in coming up with their own solutions. Once the group that you are helping completes the lab assignment, you may leave. (If it is still early, the group that received your help should then stay to help another group.)

EXPLORATIONS

If you have finished the lab exercises and are waiting to be graded, or if you wish to explore programming in Haskell further at any time, or want to get some extra practice, have a look at the Explorations page.

Exploration problems are optional and should only be attempted after other problems are completed.
You don't have to show these answers to the lab assistants for grading.