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 4

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

After this lab session you should be able to....

- Write simple recursive functions.
- Explain how recursive Haskell programs are executed.
- Write functions that generate lists.
- Give variants for recursive functions.
- Use local declarations.

Instructions

1. Read slides 1-23 from lecture 6 (Local Declarations, Patterns), all slides from lecture 7 (Recursion), and slides 1-15 from lectures 8/9 (Lists).
2. All problems in this assignments *must* be solved using recursion. Should you know of some "clever" way of solving a 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).
4. Try to give a variant for each recursive function that you write.
5. Remember to *test* your functions. Where we have given examples in the assignments, make sure that your programs can run them correctly.
6. The answers to all questions should be *written down* - preferably in a file!

The Task

PROBLEM 1

Write a function

```
stringOfInteger :: Integer -> String
```

that translates an integer into a string. For instance, if the argument is 42, the function should return the string "42". If the argument is negative (e.g., -42), the function should return a string that begins with "-" (e.g., "-42"). Use recursion. (Do *no*t use library functions, such as Haskell's show.)

Hints: You have already thought about an algorithm for this problem in the last lab. There are 10 base cases, namely any number from 0 to 9. If the argument x is negative, consider -x instead. If x>9, for the recursive call, consider how computing the values `x `div` 10` and `x `mod` 10` might be useful.

PROBLEM 2

Write a function

```
searchString :: String -> String -> Integer
searchString mainstring substring = ...
```

that takes two string arguments *mainstring* and *substring* and examines if *substring* occurs as a substring of *mainstring*. If so, the function should return the position in *mainstring* where *substring* starts. (If *substring* occurs multiple times, the position of the first occurrence should be returned.) If *substring* does not occur in *mainstring* the function should return -1.

Example:

- searchString "Jultomte" "tomte" == 3
- searchString "Jultomte" "Jul" == 0
- searchString "Jultomte" "om" == 4
- searchString "Jultomte" "gran" == (-1)
- searchString "Jultomte" "tomtenisse" == (-1)
- searchString "Kalles farfar" "far" == 7

Hints: Consider solving a more general problem, using an auxiliary function searchStringAux main sub position, where position indicates how many characters of mainstring you have searched already. What is the base case? What is the recursion scheme? Remember that we have seen several functions that return some part of a string (e.g., take, tail), and use them as necessary.

PROBLEM 3

Try to do this assignment without running the code, as an exercise in reading Haskell code. If you cannot even guess, run the code in Haskell to try and figure out what it does. Examine the function declaration below:

```
severian vodalus =
  if vodalus == ""
    then 0
    else severian (take (length vodalus - 1) vodalus) * 2 +
      if drop (length vodalus - 1) vodalus == "0"
```

```
    then 0
    else 1
```

1. What is the type of vodalus? What is the type of severian?
2. Show, *step by step*, how the expression severian "10" is evaluated. Use the same format that has been used for evaluating expressions in the lectures. (If the same expression appears several times, you need to show its step-by-step evaluation only once.)
3. Describe, *in words*, *how* severian does what it does, i.e., how the code works. Present this description at a higher level of abstraction than the code itself, and not merely for the particular example where vodalus is "10", but in general.
4. Explain, in words, *what* severian computes - that is, what *result* it computes, not how it does so. In other words, provide its postcondition.
5. Give the function severian and the argument vodalus more descriptive names.

PROBLEM 4

Write a function myReplicate :: Integer -> a -> [a] such that myReplicate n x returns the list that consists of n copies of x (where n>=0). For example:

- myReplicate 5 42 == [42,42,42,42,42]
- myReplicate 3 "!" == ["!","!","!"]
- myReplicate 0 True == []

Use recursion. (The Haskell Prelude already provides a similar function replicate. Do *not* use this function to solve this problem.)

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.

Find a variant that proves termination of myReplicate n x for all arguments n>=0.

PROBLEM 5

Write a function fromTo :: Integer -> Integer -> [Integer] such that fromTo low high returns the list [low, low+1, ..., high]. For example:

- fromTo 0 6 == [0,1,2,3,4,5,6]
- fromTo (-3) 1 == [-3,-2,-1,0,1]
- fromTo 9 7 == []

Use recursion. (Haskell provides other ways to obtain lists of the form [a, a+1, ..., b], e.g., *ranges*. Do *not* use them to solve this problem.)

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.

Find a variant that proves termination of fromTo low high for all integer arguments low, high.

PROBLEM 6

In Problem 2 above, you constructed a function searchString that used an auxiliary function. Rewrite searchString so that the auxiliary function is declared locally.

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.