Recommended exercises with hints to Chapters 5-7

– D7012E Declarative programming –

Håkan Jonsson

March 27, 2018

1 How to work

Do as explained before, that is sit by a computer with book, pen, and paper. Type in solutions and run them but also write down your code on paper to practice for the written test.

2 Chapter 5

Exercise 5.2 on page 77

I would use pattern matching, that binds names to each of the components in the argument triplet, compare the values of the components in guards, and then simply return rearranged triples as needed.

I would not bother with all those functions mentioned.

Exercise 5.10 on page 82

1. Assume we are to write `divisor n`, so `n` is the name of the parameter. Then we can easily solve this with a list comprehension involving the list of all numbers from 1 to n. You like to have "the list of all numbers x such that x belongs to the list of numbers from 1 to n and x divides n".

2. Just check what `divides` returns.

Exercise 5.11 on page 82-83

1. Also, `matches` is easy to implement with a list comprehension. Assume that the parameters are names k, for the integer, and xs, for the list. Then we like to have "the list of all numbers x such that x is an element of xs and $k = x$".

2. `elem` just needs to check the length of what `matches` returns. (Don't bother hiding things from `Prelude`. Instead, use another name.)

Exercise 5.18 on page 90

Without the type signature row, the types become polymorphic with type variables instead of `Int`.

Write the type down on paper, as you think it will be. Then program the function `shift` as given and check its type with the command

```
:type shift
```

Exercise 5.22 on page 94

The trick is to include the character `'\n'` in the string where a print out should continue on the next row.

Use `putStr (onSeparateLines <your list>)` and verify that the output is really broken up.

Exercise 5.23 on page 94

Thinking recursively, the function will consist of two cases: A base case and a recursive case. Use the template I have shown.

Exercise 5.24 on page 94

Better is to implement the function `pushRight :: Int -> String -> String`, where the integer takes the role `linelength` has in the original exercise.

You need to be able to generate strings with blanks and of certain lengths.

(This function could be useful in lab assignment 1.)

3 Chapter 6

Exercise 6.29 on page 112-113

Do exercise 7.2 before this one.

The interesting part of this exercise is how to count the discounts. Skip the functions asked for.

I would sort the list (bar codes are integers) and then go though the sorted list to identify the number of identical bar codes that lie next to each other. However, I then

must be careful so I do not count three identical in a row as two pairs. Three in a row would be one pair and one additional.

To do this, my recursive function needs to be defined for different cases. The most important one is where the argument matches the pattern `x1 : x2 : xs`, a list with head `x1` followed by the element `x2` (and then then `xs` the rest of the list).

4 Chapter 7

Exercise 7.2 on page 119

Basic pattern matching of lists. Easiest solution is a function with three cases mathcing a) empty lists, b) lists with just one element, and c) lists with at least two elements.

Exercise 7.3 on page 119

Rewrite just 7.2 (not 7.1) as a function with just one case that matches the argument list with a name `xs`[1]. Examine the list with functions like `length :: [a] -> Int`, `head :: [b] -> b`, and `tail :: [b] -> [b]`.

Exercise 7.4 on page 120

First define it as a linear recursive function with a base case and a recursive case.

Can you define it also with `foldr`? Compare with how we can write `sum` with `foldr`, a function that adds, and 0 (the result in the base case).

Exercise 7.5 on page 120

Same help as in Exercise 7.4, but here the operations are boolean. Using `foldr`, the answers become (very) short "one-liners".

Exercise 7.7 on page 125

We need a local helper function that can remove all occurences of a value from a list. This can be done with a list comprehension. Or we could do it with a recursive function we write.

Exercise 7.8 on page 125

Use the template for linear recursive functions. Note the types:

- `reverse :: [t] -> [t]`
- `unzip :: [(a,b)] -> ([a],[b])`, turns a list och pairs into a pair of lists.

[1] In the book they do not consider this pattern matching but I do so.

Exercise 7.9 on page 125

Assuming `iSort` has been written, it is possible to extract what we need from its result since it is the first and the last elements.

Read through the list of functions in Section 5.8 and you will find something useful.

Exercise 7.14 on page 128

The definition of `take` can be found on page 127. `take i xs` returns a list with the first `i` elements of `xs` (in the same order).

`drop i xs` returns the list with everything in `xs` except `take i xs`, that is

```
xs = take i xs ++ drop i xs.
```

Exercise 7.18 on page 133

NB! The term "sublist" is used differently in lab 1 then in this exercise!!! Read carefully!!! This is a *very fine* exercise that only those that know recursion by heart can solve.

- Non-empty lists are not sublists of empty lists but empty lists are sublists of non-empty lists. A list `x:xs` is a sublist of a list `y:ys` if either `x==y` and `xs` is a sublist of `ys` or `x:xs` is a sublist of `ys`.
- Non-empty lists are not subsequences of empty lists but empty lists are subsequences of non-empty lists. A list `xs` is a subsequence of a list `y:ys` if `xs` is a prefix of `y:ys` or `xs` is a subsequence of `ys`.

  Non-empty lists are not prefixes of empty lists but empty lists are prefixes of non-empty lists. A list `x:xs` is a prefix of a list `y:ys` if `x==y` and `xs` is a prefix of `ys`.

Exercise 7.19-7.23 on page 129-133 (harder)

On page 129, an example of how text processing might be done is given. Read it, program all the functions (type them) and run them. Get to know them.

The most interesting exercise is 7.23, where we are to *justify* text. This means the left and right margin should be straight. To achieve this we need to split the text and insert whitespace between words on rows. The total number of charcters (including white-space) on a row should be, say, n.

The input, bascically a long string, must first be hacked up into a list of words (including separating characters like commas and periods). Then, we need to go though the words in order and fill rows as much as possible, that is we must make sure no row becomes longer than n.

I would use a list of strings as a temporary representation of a row. The list would hold the words and between the words I would put a string contining just one white-space

to start with. A row with k words will then contain $2k - 1$ strings; the odd-numbered ones being words and the even-numbered ones being short white-space strings.

Now, rows in which the combined length of all strings is less than n characters need to be made longer. This is done by adding white-space to the white-space strings (not the words). From the examples given in the book, we can see that white-space is added "evenly" to all white-space strings, and to the extent possible, to spread out all words evenly on the row. If the number of white-space characters that need to be added is not exactly a multiple of the number of white-space strings, we rather add to the left-most strings than the right-most ones. In this case, the spacing between words will be larger to the left on the row.

The temporary rows, our lists of strings, are then concatenated into real rows and returned as the final result.

Exercise 7.25 on page 133

First, note that we only care about letters and that no distinction is made between capital and small letters. You need to filter out non-letters and change so all letters are either capital or alla are small.

In a palindrome, the first and last characters are equal, and the list between those characters is a palindrome.

Or, a list `xs` is a palindrome if it reads the same as `xs` reversed.

Exercise 7.26 on page 133 (harder..??)

This problem might seem hard but it is straight-forward. Reuse the function that checks if a list is a prefix of another list from Exercise 7.18. That function, together with the built-in function `drop`, can be used to write a simple recursive function with two cases; one where what we like to substitute is the prefix and one where it is not. That's it(!)