

Recommended exercises with hints to Chapter 17

– D7012E Declarative programming –

Håkan Jonsson

April 12, 2018

1 How to work

- Sit by a computer with your book, pen, and paper.
- Work on the exercises below one at a time and in order.

2 Chapter 17

Exercise 17.1 on page 349

First do Exercise 17.6. Then, like with all step-wise evaluations, use the results of 17.6 and write rows that show what happens between the steps.

Exercise 17.2 on page 350

Be careful with the definitions, that might be different from what you have read elsewhere. You have by now probably already written `subsequence`, right? But most likely given it another name.

Regarding the general case of `subList`, where the argument is a non-empty list `x:xs`, can be thought of like this. The recursion assumption will be a list with all sublists of `xs`. Now, these sublists are also sublists of `x:xs`. However, for each such sublists we get another sublist by prepending `x` to it.

Exercise 17.4 on page 350

Recall that `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` will go through both lists and use the function on pairs of elements to compute the elements of the list that is returned. We also need to sum up the returned list, with `foldr` for instance.

Exercise 17.22 on page 369

We have done the fibonacci sequence in class. As for the sequence of factorial numbers, we could start with all natural numbers $0, 1, 2, \dots$. For each number i we create the list of numbers $1, 2, \dots, i$ and just multiply them all together. With `foldr` and `map`, this becomes a one-liner.

Exercise 17.23 on page 369-370

Start by writing `factors`. Easiest is to generate the list of numbers $1, 2, 3, \dots, n$ and then filter out just those that divide n evenly.

Now, the list `hamming :: [Int]` can be generated from `[1..]` by filtering and only keeping those numbers i that has as one of its factors 2, 3, or 5. And add 1 in the front. Done.

Exercise 17.24 on page 370

The running sum is the sum of two lists both of which are based on the argument list. Which ones? Find out and then just `zipWith` them together by adding elements(!)

Exercise 17.27 on page 373

Having solved 17.24 we just need to filter the argument list before we use it in the running sum. I suggest writing a more general function

```
runSum :: (Int -> Bool) -> [Int] -> [Int]
```

that only computes a running sum of the integers from the list that the first argument (a function with type `Int -> Bool`) returns `True` for. Then, we get the function asked for in 17.27 as `runSum (>=0)` but can also easily put other constraints on the input.

Exercise 17.28 on page 373

This exercise is simpler than one might think. Do as you would do if the lists were finite and skip the base cases(!)