Recommended exercises with hints to Chapters 9-10

– D7012E Declarative programming –

Håkan Jonsson
Luleå University of Technology, Sweden

April 3, 2019

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 9

Exercise 9.2 on page 159

Note that `foldr (+) 0` is a function that sums up numbers in a list. It definitely works on, for instance, lists with ones.

Exercise 9.4 on page 160

Not really a programming exercise but still valuable. Try to write

```
map f (map g xs)
```

using just one `map`.

Exercise 9.6 on page 160

This is an exercise in how to use `map` and `filter`. (Since these two can be defined with list comprehensions, it is also possible to solve the exercise with list comprehensions. You might like to write solutions using both ways.)

Exercise 9.7 on page 160

The last one is a bit tricky but what if we could `zip` together `f` applied to `[0..n-1]` and `f` applied to `[1..n]`, and see how many pairs there are in which the first component is not the largest?

Exercise 9.9 on page 160-161

What we like to do is to *compose* a total of n copies of `f`. The base case might seem strange but composing 0 functions should result in a polymorphic function that just returns its argument as it is.

Exercise 9.10 on page 161

First, declare a function that doubles its argument. Then, use `iter` from Exercise 9.9. Try it out in practice. (This shows that even a strange function like `iter` can be quite handy.)

Exercise 9.11 on page 163

If a list comprehension gives the list of numbers, `map` and a suitable lambda expression can be used to square the numbers. And then use the hint I wrote to Exercise 9.2.

Exercise 9.16 on page 164

A plain recursive function with some cases solves the problem.

Exercise 9.17 on page 164

Use `filterFirst` and something that reverses lists.

3 Chapter 10

Exercise 10.3 on page 171

This exercise is like Exercise 9.9 but instead of getting a function and a number, we get a list `ls` with the functions the should be composed.

1. All elements of a list must have the same type, so what type must the functions in `ls` have [to be composable]?

2. If you understand how `foldr` works, the solution to this exercise becomes a short "one-liner".

Exercise 10.7 on page 175

First: Since there is a function `flip` in `Prelude`, use another name (`flip2`, for instance).

This is an example of a function of higher order that "changes a function". Now, nothing in Haskell can change so what it really does is to return a *new* function that does the same job as the argument function (whose type is `a -> b -> c`) but that takes its arguments in the other order (`b -> a -> c`). The type is the best hint to how to solve this exercise.

(The solution to this problem resembels something you have just read, is surprisingly short, natural, and – when you have seen it – so obvious.)

Exercise 10.8 on page 175

This is an exercise to formulate a lambda expression. The functions we should use have the types

```
not :: Bool -> Bool
elem :: Eq t => t -> [t] -> Bool
```

and then we should not forget the little string `" \t\n"`.

Exercise 10.13 on page 180

Study some examples. Try what `filter (>0) . map (+1)` applied on some different lists gives.

Exercise 10.14 on page 183

Let's make a small change: Squares consist of *two* characters. This makes the print-outs much nicer (on most displays). Then, an 8-by-8 chess board looks like this:

```
  ##  ##  ##  ##
##  ##  ##  ##
  ##  ##  ##  ##
##  ##  ##  ##
  ##  ##  ##  ##
##  ##  ##  ##
  ##  ##  ##  ##
##  ##  ##  ##
```

Note that it consists of two kinds of rows that alternate, and each row consists of an alternating sequence of a) two white-space characters and b) two hash marks (`##`). Write functions that returns

1. a white square (a string with two white-space characters),
2. a black square (two hash marks)
3. two squares (a string with four characters), a white followed by a black,
4. two squares, a black followed by a white,
5. a row (that is, a string terminated by a `'\n'`) with n squares in which the first is white,
6. a row with n squares in which the first is black, and (finally)
7. an n-by-n chess board.

You could also try to merge functions above and compute the final function with just a few help functions.

Exercise 10.20-10.32 on page 191-193

(These exercises all have to do with the example that makes up the entire Section 10.8. I recommend looking at them but note that they take considerable time since they rely on first having understod the details of the example.)

Exercise 10.22 on page 191-192

Write the function `compact :: [Int] -> String` that compacts ranges in *an ordered* list of integers and creates a string [that could be on a line in an index of a book]. Example:

```
compact [1, 3, 4, 5, 7, 10, 11, 18, 20]
```

should be

```
"1, 3-5, 7, 10-11, 18, 20".
```

You might like to start by constructing a list of integer pairs with one pair for each integer in the list. The list `[1, 3, 4]` would yield the list `[(1,1), (3,3), (4,4)]`. The pairs represent ranges, all of which are just one to start with. And then you go through the list of pairs recursively and examine the first pair and the first pair in the recursive assumption in the recursive step. The final step is to turn all those pairs into a string, perhaps a job for `foldr`! (Or a linear recursive function.)