Recommended exercises with hints to Chapter 14

– D7012E Declarative programming –

Håkan Jonsson
Luleå University of Technology, Sweden

April 8, 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 14

Exercise 14.1 on page 248

Function `weather` can be found on page 243, where it is defined as a 2-case function. Re-define it too instead be a 1-case function using either

- guarded equations (lines with `| <guard> = <result>` under the first line of the function) or
- one conditional expression (`if`).

Exercise 14.4 on page 248

Use `Shape` on page 247, where instances for the type are automatically derived in classes `Eq`, `Show`, `Read`, and `Ord`. Deriving instances like this is very practical.

Your function – let's name it `perimeter` – needs two cases, one per kind of shape there is. As for the type, try to figure it out by studying the argument and the result. You could also let Haskell tell you the type of your function.

Exercise 14.5 on page 248

A triangle is defined by its side lengths.

`isRound` (on page 246) and `perimeter` are straight-forward to program. The area is given by *Heron's formula* and the function `area` to change can be found next to `isRound`.

Exercise 14.6 on page 248

Let's name the function `isRegular`. It will consist of three cases, one per kind of `Shape`, and just return `True` (for circles) or the result of boolean expressions that compare sides (for the other two shapes). No need to make it more complicated.

Exercise 14.8 on page 248

Write a more general `instance` declaration for `Eq Shape` that makes not only 1) shapes based on identical numbers equal but also 2) shapes of the same kind defined by one or several negative numbers equal(!) The latter means that two shapes *of the same kind* are equal if at least one number in each of them is negative. In case a shape is defined by more than one number (like rectangles and triangles), the numbers that are negative do not need to be at the same positions.

Exercise 14.9+14.10 on page 248-249

You have two options: 1) Change each kind of shape to include the center or 2) make a totally new type with two separate parts: A shape and a (center) point.

Either way, start by defining an algebraic type `Point` for a point in the plane. Then, call the new type `NewShape` as suggested.

1) Copy constructors from `Shape`. Add to each a `Point`.

2) Find a suitable name for a shape that has a position. This will be the name of the constructor. Then, declare `NewShape` as the product type of `Shape` and `Point`. (Note that a `Shape` has no center.)

The function `move :: Float -> Float -> NewShape -> NewShape` could use pattern matching to take apart the `NewShape` n given as an argument to it, and then return a new `NewShape` with a center computed from n and the two other arguments.

Exercise 14.15 on page 255

NB! These types are similar to those in Labs 2 and 3 but not the same!

Write down the reduction steps and show how the initial expression, one step at a time, turns into the final answer. When in doubt, start to the left.

Exercise 14.16 on page 255

NB! These types are similar to those in Labs 2 and 3 but not the same!

This will be a recursive function with cases for all the possible expressions there are. We basically count constructors other than `Lit`.

Exercise 14.17 on page 255

Change the type to include also **mul**tiplication and integer `div`ision but then it is enough to change `eval`.

Handle division by zero by calling the built-in function `error`; see page 261.

Exercise 14.18 on page 255

Create a new type `Expr2` in this new way (base it on `Expr` but save `Expr` as it is to later).

It is enough to change `eval`. You need the same (new) patterns in all of the functions.

Exercise 14.21 on page 256

An `NTree` (page 251) is a type for binary trees with integers in the nodes (not the leaves). On page 252, functions `sumTree`, `depth`, and `occurs` are typical examples of recursive functions over binary trees.

Our functions need to use pattern matching. What should happen if the tree is just a leaf?

Exercise 14.22 on page 256

The function – I call it `elem` – will have the type `elem :: Int -> NTree -> Bool`. Just recur over the two kinds of trees there are.

Exercise 14.23 on page 256

This exercise is interesting because it requires another base case in addition to a leaf.

If the tree is just a leaf, we have an error condition. If not, we find the maximum and minimum in the subtrees. These values and the value in the node gives us the maximum and minimum we should return.

But what extra base case is needed..?

Exercise 14.24 on page 256

Again, the solution is a simple (branching) recursive function. Just take apart the argument (tree), reflect the subtrees, assemble the reflected trees into the result. Reflecting a leaf gives the (same) leaf.

Exercise 14.28 on page 260

Just change the function asked for in 14.22 so it instead works for values of type `Tree a` (page 258). This makes it polymorphic, and we need to add a suitable type constaint (`... =>`) so the values in the tree can be compared with `==`.

Exercise 14.29** on page PPP

`twist` comes with ... a twist: If the first kind, return the second; if the second, return the first. This insight, that make the function very simple (just two cases), effectively "turns the type variables around".

Exercises 14.30-14.32 build upon 14.29 but are not essential for us, so do them only if you are very interested in abstracting functions.

Exercise 14.33 on page 260

Good exercise with 6 parts resulting in recursive functions handling

```
data GTree a = Leaf a | Gnode [GTree a]
```

One way or another, all functions traverse the list in a `Gnode`. Try to solve all of them using `foldr` and (if needed) `map` – or, if you like a challenge: Use `foldr` to also perform the tasks carried out by `map`(!) In either case, refrain from defining recursive functions using the template shown in class.

countGT I would have a base case that returns 1 for trees that are leaves. In general, we map `countGT` on the list in the `Gnode`, and them sum up the numbers in resulting list. (Computing the sum of numbers in a list can be done with `foldr`, in an expression with just 11 characters.)

depthGT A leaf has depth 1 (or 0). Otherwise, we take the maximum of the depths of all trees in the list in the `Gnode`. Perhaps the predefined function `max` can be of help?

sumGT The code of this function is very similar to the code of `countGT`.

elemGT It is simple to check if a leaf contains the value we are looking for. In general, a value `x` is conatined in one of the trees in a list of trees `[t1, t2, ..., tn]` if, and only if, `x`$\in$`t1` or `x`$\in$`t2` or ... or `x`$\in$`tn`.

mapGT This function "changes" the leafs only. Given a leaf, this change is easy to do: Just return a new leaf with the changed value. In general, we need to run `mapGT` on all elements in the list of the `Gnode`, and then construct a new `Gnode` of the resulting list.

flattenGT Here, in general, we like to map `flattenGT` on the list of tress, which gives a list of lists, and then we append, perhaps using `(++)`, all these lists into one.

Exercise 14.34 on page 260

By “empty” is meant “does not contain any values”.