CS-205 Declarative Programming — October 31st 2022
Cécilia Pradic, Monika Seisenberger

Lab 5: algebraic datatypes

Dear all,

- You can do your lab by editing a single file.
- If you want to sign off, please recall to load your files in `ghci` to check for errors before calling a demonstator over. Additionally, it is also a good idea to test the functions you defined in `ghci` on a variety of examples to double-check they have the behaviour you are expecting.

The goal is to get familiar with custom datatype definitions as we have seen in the last two lectures. But this also means we will revisit material on recursion and higher-order functions among other things!

Happy hacking!

Task 5.1. A simple datatype

(i) Define a custom datatype enumerating the four cardinal directions. Your definition should make the following function definition valid:

```haskell
turnAround :: Direction -> Direction
turnAround North = South
turnAround South = North
turnAround West = East
turnAround East = West
```

(ii) Define a function `turnLeft :: Direction -> Direction` that takes a direction to the one which is leftwards. Analogously, define the inverse function `turnRight`.

Task 5.2. Using the Maybe datatype Recall the `Maybe` type constructor that we have seen in the lectures. It is defined as follows in Haskell's Prelude (**do not copy-paste the following snippet in your file, it is normal that doing so would cause you issues later on**):

```haskell
data Maybe a = Just a | Nothing
```

`Maybe` is often useful if one wants to define functions that may return an error. For instance, if we would like to have an error-free version of `head`, one could write

```haskell
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead (x : _) = Just x
```

Taking inspiration from this example, write the following:

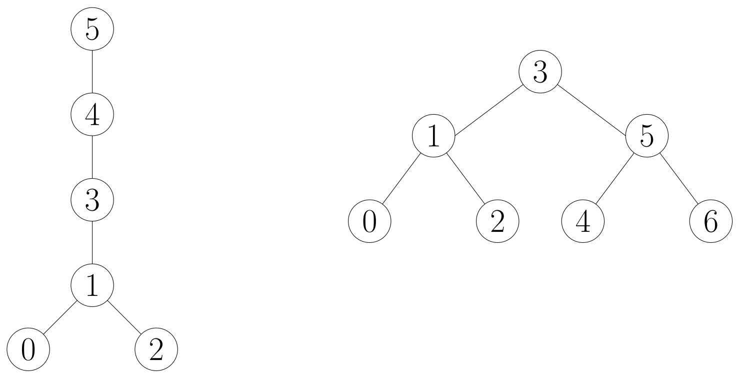

Figure 1: Some informal pictures of `ex1 :: Btree Int` and `ex2 :: Btree Int`

(i) a function `safetailMaybe :: [a] -> Maybe [a]` that returns `Nothing` on the empty list and returns the correct value wrapped by `Just` if the list is non-empty

(ii) a function

```
takeMaybe :: Int -> [a] -> Maybe [a]
```

that returns `Nothing` in case that the first input is strictly greater than the length of the second, and otherwise use `Just` to give a result similar to `take`.

Task 5.3. A recursive tree datatype We will now consider a recursive datatype for labelled binary trees that you should copy-paste in your file:

```
data Btree a = Leaf a | Unary (Btree a) a | Binary (Btree a) a (Btree a)
```

Per that definition, a value of type `Btree a` is either:

- a leaf node containing a value of type `a`
- or a unary internal node containing a subtree of type `Btree a` and a value of type `a`
- or a binary node containing two subtrees of type `Btree a` and a value of type `a`

Here are a couple of concrete examples of values of type `Btree Int`:

```
ex1 = Unary (Unary (Unary (Binary (Leaf 0) 1 (Leaf 2)) 3) 4) 5
ex2 = Binary (Binary (Leaf 0) 1 (Leaf 2)) 3 (Binary (Leaf 4) 5 (Leaf 6))
```

which are also pictued in Figure 1 for your convenience.

We encourage you to test your functions on these examples to check if they work!

(i) Define a function

```
depth :: Btree a -> Int
```

computing the *depth* of the tree, that is, the maximal distance between a leaf and the root (so, by convention, the depth of a tree consisting of a single leaf is 0). For instance, we have `depth ex1 == 4` and `depth ex2 == 2`.

(ii) Recall the `map` function over lists, that uniformely applies a function `f` to the elements of an input list. Write an analogous function

```
mapBtree :: (a -> b) -> Btree a -> Btree b
```

such that `mapBtree f t` is the same tree as `t`, except for the labels which should be obtained by applying `f` to the labels of `t`.

To test the function, you might like to define the auxiliary function

```
btreeToList :: Btree a -> [a]
btreeToList (Leaf x)       = [x]
btreeToList (Unary l x)    = btreeToList l ++ [x]
btreeToList (Binary l x r) = btreeToList l ++ x : btreeToList r
```

and check that `map f (btreeToList t) == btreeToList (mapBtree f t)` for any values of `f` and `t` you can come up with.

Task 5.4. Challenge tasks

(i) Write a function

```
bindMaybe : Maybe a -> (a -> Maybe b) -> Maybe b
```

which allows to "chain" computations with `Maybe`: when calling `bindMaybe x f`, if the first argument `x` is `Nothing`, the output should be `Nothing`, otherwise `x` is of shape `Just y` and the output should be `f y`.

(ii) Revisit question 2-(ii) and write `takeMaybe` without reusing the function `take` from Haskell's `Prelude` (you may want to reuse `bindMaybe` for this).

(iii) A value `t` of `Btree (Int, a)` is called a *search tree* if the root of every subtreee of `t` is either a leaf or

- a unary node `Unary l (x, v)` and all values contained in `l` have first components less or equal than `x`
- or a binary node `Binary l (x,v) r` and all values contained in `l` have first components less or equal than `x`, and those contained in `r` have first components greater or equal than `x`.

Such trees are useful, because it is possible to retrieve elements by key in time $\mathcal{O}(\texttt{depth}(\texttt{t}))$. This can be logarithmic in the size of the tree if it happens to be balanced.

Write a function

```
lookupInSearchTree :: Int -> Btree (Int, a) -> Maybe a
```

that looks up an element in the tree. That is, `lookupInSearchTree k t` should return `Nothing` if there is no label with first component `k` in `t`, otherwise `Just y` where `y` is some value such that `(k,y)` occurs in the tree.

For instance, `lookupInSearchTree 5 ex5` should evaluate to `Just "c"` while `lookupInSearchTree 100 ex5` should evaluate to `Nothing`.

Then, write a function

```
insertInSearchTree :: Int -> a -> Btree (Int,a) -> Btree (Int,a)
```

such that `insertInSearchTree x v t` results in a search tree containing all the values of `t` plus one copy of `(x,v)` (you do not have to enforce that the output tree be balanced).

(iv) A tree is called *perfect*[1] if the distance between the root and any leaf is always the same and all internal nodes are binary. Neither `ex1` nor `ex2` are perfect. A tree is called *complete* if the distance between the root and any leaf is always the same up to 1, all of the deepest leaves are located to the left of the others, and there is at most one internal node with unary node that should be located on the second-to-last level. `ex2` is complete, but `ex1` is not.

Define functions

```
complete :: Btree a -> Bool
perfect :: Btree a -> Bool
```

that decide whether an input tree is complete/perfect. Please attempt to code a version of `perfect` that is efficient, i.e., than only needs to examine every node of the input tree exactly once.

[1]The definition we give are consistent with `https://en.wikipedia.org/w/index.php?title=Binary_tree&oldid=1050054313`, so you can check there for further examples.