universität innsbruck

| Functional Programming | WS 2023/2024 | LVA 703025 |
|---|---|---|
| Exercise Sheet 5, 10 points | | Deadline: Tuesday, November 14, 2023, 8pm |

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_05.hs` provided on the proseminar page.
- Upload your modified .hs file in OLAT.
- Your .hs file must be compilable with `ghci`.
- Try to define auxiliary functions within a `where` or `let ... in` construct.

## Exercise 1 *Recursion on Lists* **6 p.**

1. Define a type synonym `Age` for a tuple containing the name and `Integer` age of a person. What is the difference between the keywords `type` and `data` in Haskell? (0.5 points)
   **Examples:**
   ```
   exampleAges :: [Age]
   exampleAges = [("Alice",17), ("Bob",35), ("Clara",17)]
   ```

2. A ticket costs €5 for a child aged 0–12, €7.50 for a teenager aged 13–17, and €15 for an adult aged $\geq 18$. In this task, you will implement two equivalent functions `ticketCostA`, `ticketCostB :: Age -> String` which return a string `"[name] pays [cost] euros for a ticket"` using different Haskell constructs. To avoid copy-pasting strings, define a *local* auxiliary function `formatCost :: String -> String` which takes a cost and returns the output string for each variant. (1.5 points)
   (i) Implement `ticketCostA` using if-then-else expressions to differentiate between ages. Define the auxiliary function `formatCost` using a `let`-expression. You may not use guarded equations.
   (ii) Implement `ticketCostB` using guarded equations. Define the auxiliary function `formatCost` using a `where`-construct. You may not use any if-then-else expression.

   **Examples:**
   ```
   ticketCostA ("Alice",17) == "Alice pays 7.50 euros for a ticket"
   ticketCostB ("Bob",-1)    -- Causes a sensible error
   ```

3. Write a function `ageLookup :: [Age] -> Integer -> Maybe [String]` which takes a list of ages and a specific age. If there is at least one person with this age, then a list of the names of people with this age should be returned, otherwise `Nothing` should be returned. (1.5 points)
   *Hint: you might need a recursive call of* `ageLookup`*. Try using a* `case ... of ...` *to differentiate between the* `Just` *and* `Nothing` *cases rather than writing a separate auxiliary function.*
   **Examples:**
   ```
   ageLookup exampleAges 17 == Just ["Alice", "Clara"]
   ageLookup exampleAges 10 == Nothing
   ```

4. Implement a Haskell function `bidirectionalLookup:: [(a, b)] -> Either a b -> Maybe (Either a b)` that takes a list of pairs of type `[(a,b)]` and a key `k :: Either a b`. For keys of shape `Left l`, perform a lookup on the left half of the pairs, and for keys `Right r` on the right half of the pairs. In both cases, return the other half of the first matching pair. If no match is found, the function should return `Nothing`. (2.5 points)

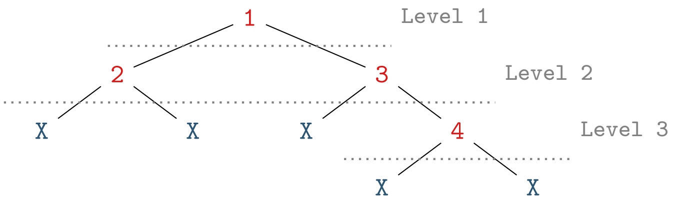
**Examples:**
```
bidirectionalLookup exampleAges (Left "Bob") == Just (Right 35)
bidirectionalLookup exampleAges (Right 17) == Just (Left "Alice")
bidirectionalLookup exampleAges (Right 10) == Nothing
```

## Exercise 2 *Combined Recursion* **4 p.**

Consider the following data type for binary trees:

```
data Tree a = Node a (Tree a) (Tree a) | X deriving Show
```

The data type is similar to the one from Sheet 04 but with a constructor `X` instead of `Leaf` to represent an empty tree. For example, `exampleTree` from `template_05.hs` represents the following tree




where the different *levels* of the tree are indicated by dotted gray lines.

1. Implement a function `takeLevels :: Int -> Tree a -> Tree a` such that `takeLevels n t` results in a tree consisting of the upper `n` levels of the tree `t`. (1 point)
   **Examples:**
   ```
   takeLevels 2 exampleTree == Node 1 (Node 2 X X) (Node 3 X X)
   takeLevels 0 exampleTree == X
   ```

2. Implement a function `dropLevels :: Int -> Tree a -> Tree a` such that `dropLevels n t` results in a list of trees consisting of the subtrees that remain after removing the `Node`s of the upper `n` levels of the tree `t`. Since only `Node`s are removed, `X`s "hanging on" removed `Node`s should "fall down," which is achieved by fixing the equation `dropLevels _ X = [X]`. (1 point)

   **Examples:**
   ```
   dropLevels 2 exampleTree == [X, X, X, Node 4 X X]
   dropLevels 0 exampleTree == [exampleTree]
   ```

3. Without using `takeLevels` and `dropLevels` from above, implement a function

   ```
   splitAtLevel :: Int -> Tree a -> (Tree a, [Tree a])
   ```

   that combines the functionality of `takeLevels` and `dropLevels` into a single recursive function. (1 point)

   *Hint:* look at the similarities in the recursive structure of `takeLevels` and `dropLevels` and combine what you find using pattern matching on tuples.
   **Example:**
   ```
   splitAtLevel 2 exampleTree == (Node 1 (Node 2 X X) (Node 3 X X), [X,X,X,Node 4 X X])
   ```

4. Implement a function `fillXs :: Tree a -> [Tree a] -> (Tree a, [Tree a])` such that `fillXs t ts` uses the trees from the list `ts` to fill-in the `X`s in the tree `t` and returns this result together with the remaining trees of `ts` that did not replace any `X`s. (1 point)

   *Hint:* Whenever `splitAtLevel i t == (s, ss)`, then the implementation should satisfy the equation `fillXs s ss == (t, [])`.
   **Example:**
   ```
   fillXs (Node 1 X X) [Node 2 X X, Node 3 X X, Node 4 X X] ==
       Node 1 (Node 2 X X) (Node 3 X X), [Node 4 X X])
   ```