Programming Paradigms
Practical session, Week 4

Department of Informatics
University of Beira Interior

Chapter 7 slides

1. Show how the list comprehension

   ```
   [f x | x <- xs, p x]
   ```

   can be re-expressed using the higher-order functions `map` and `filter`.

2. Define `reverse`, which reverses a list, using `foldr`.

3. Without looking at the standard prelude, define the following higher-order library functions on lists.

   (a) Decide if all elements of a list satisfy a predicate:

   ```
   all :: (a -> Bool) -> [Bool] -> Bool
   ```

   (b) Select elements from a list while they satisfy a predicate:

   ```
   takeWhile :: (a -> Bool) -> [a] -> [a]
   ```

4. Using `map` and `filter` define a function that returns the sum of the squares of the even integers from a list:

   ```
   sumsqreven :: [Int] ->Int
   ```

5. Redefine the functions `map f` and `filter p` using foldr.

6. Using foldl, define a function

   ```
   dec2int :: [Int] -> Int
   ```

   that converts a decimal number into an integer. For example:

   ```
   >dec2int [2,3,4,5]
   2345
   ```

7. Without looking at the definitions from the standard prelude, define the higher-order library function `curry` that converts a function on pairs into a curried function. Hint: first write down the type of the function.

8. Define a function

   ```
   altMap:: (a -> b) -> (a -> b) -> [a] ->[b]
   ```

   that alternately applies its two argument functions to successive elements in a list. For example:

   ```
   >altMap (+10) (+100) [0,1,2,3,4]
   [10,101,12,103,14]
   ```

9. Using `foldr` define a function `sumsq` which takes an integer n as its argument and returns the sum of the squares of the first n integers, i.e.:

   $$sumsq\ n \ = \ 1^2 + 2^2 + ... + n^2$$

   Do not use the function `map`.

Chapter 8 slides

1. Consider the following type:

   ```
   type Assoc k v = [(k,v)]
   ```

   which associates a key with a value. Define a function

   ```
   find :: Eq k => k -> Assoc k v -> v
   ```

   that returns the first value that is associated with a given key.

2. In a similar manner to the function `add` (see slides), define a recursive multiplication function

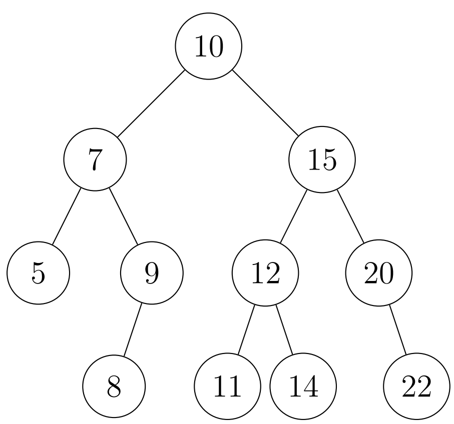
   ```
   mult :: Nat -> Nat -> Nat
   ```

   for the recursive type of natural numbers.
   Hint: make use of `add` in your definition.

3. A data type for binary trees can be defined as

   ```
   data Tree a = Leaf a | Node (Tree a) a (Tree a)
   ```

   Binary search trees are a particular type of tree that keep the values in their nodes in sorted order so that lookup and other operations can use this order to decide if the left or right subtree should be explored. The following is an example of a binary search tree:



   Using guarded equations define the function

   ```
   occurs :: Ord a => a -> Tree a -> Bool
   ```

   that decides if a given value occurs in a tree. It must do one single lookup through the tree (i.e. at each step it only searches the relevant subtree).

4. The standard prelude defines the type

   ```
   data Ordering = LT | EQ | GT
   ```

   together with a function

   ```
   compare :: Ord a => a -> a -> Ordering
   ```

   that decides if one value in an ordered type is less than (LT), equal to (EQ), or greater than (GT) another value. Using this function, redefine the function

   ```
   occurs :: Ord a => a -> Tree a -> Bool
   ```

   for search trees. Why is this new definition more efficient than the original version?

5. A binary search tree is balanced if the number of leaves in the left and right subtree of every node differs by at most one, with leaves themselves being trivially balanced. Define a function

   ```
   balanced :: Tree a -> Bool
   ```

   that decides if a binary tree is balanced or not.
   Hint: first define a function that returns the number of leaves in a tree.

6. Given the type declaration data

   ```
   data Expr = Val Int | Add Expr Expr
   ```

   define a higher-order function

   ```
   folde :: (Int -> a) -> (a -> a -> a) -> Expr ->a
   ```

   such that folde f g replaces each `Val` constructor in an expression by the function f, and each `Add` constructor by the function g.

7. Using folde, define a function

   ```
   eval :: Expr ->Int
   ```

   that evaluates an expression to an integer value, and a function

   ```
   size :: Expr ->Int
   ```

   that calculates the number of values in an expression.

8. Complete the following instance declarations:

```
instance Eq a =>Eq (Maybe a) where
...
```

and

```
instance Eq a =>Eq [a] where
...
```