Programming Paradigms 2022
Session 7: Catching up

Problems for solving and discussing

Hans Hüttel

25 October 2022

About this session

This time we will be re-visiting problems from previous session that either caused headaches or were numbered problems that we never got to solve because we ran out of time.

The overall goal this time is to help you better master topics from the first half of the course, and in particular those of recursion and datatypes.

Problems that we will definitely talk about

1. ***(Everyone at the table together – 10 minutes)***

   A former minister of science and education now wants to get a masters degree and is learning Haskell. The minister is trying to construct a function triples that takes a list of tuples (each tuple has exactly 3 elements) and converts that list of tuples into a tuple of lists.

   triples [(1,2,3) , (4, 5, 6), (7, 8, 9)] should produce ([1,4,7], [2, 5, 8], [3, 6, 9]).

   The minister wrote the following piece of code and a type specification but ran into problems. What seems to be wrong?

   ```
     triples :: Num a => [(a,a,a)] -> ([a],[a],[a])

   triples [] = ()
   triples [(a,b,c)]= ([a],[b],[c])
   triples (x:xs,y:ys,z:zs) = [x,y,z] : Triples [(xs,ys,zs)]
   ```

2. ***(Everyone at the table together – 20 minutes)***

   Now fix the issues that the triples function has. Write a definition of triples that actually works. *Please use recursion and local declarations in your solution and use the approach described in Section 6.6.*

3. ***(Work in pairs– 20 minutes)***

   The function wrapup is a function that takes a list and returns a list of lists. Each list in this list contains the successive elements from the original list that are identical.

   For instance, wrapup [1,1,1,2,3,3,2] should give us the list [[1,1,1],[2],[3,3],[2]] and wrapup [True,True,False,False,False,True] should give us the list [[True,True],[False,False,False],[True]].

   Give a *recursive* definition of wrapup in Haskell. First try to find out what the type of wrapup should be and follow the overall approach described in Section 6.6. *Hint:* Please use local declarations!

4. ***(Everyone at the table together – 20 minutes)***

   On page 98, the book describes search trees; make sure that you understand what important property a search tree has.

   Assume that our type for trees is defined as

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

   This means that trees can now also be empty. Define a function

```
insert :: Ord a => Tree a -> a -> Tree a
```

such that whenever t is a search tree, then insert t x gives us a new search tree that now also contains x.

More problems to solve at your own pace

In your solutions, remember the learning goals of this session!

a) We say that a binary tree is *balanced* if the number of leaves in every left and right subtree differ by at most one with leaves themselves being trivially balanced. Define a function balanced that will tell us if a binary tree is balanced or not. *Hint:* It is a good idea to first define a function that finds the number of leaves in a tree.

b) On page 106 there is a definition of the Expr datatype for expressions. Define a higher-order function

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

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

Then use foldexp f g (with appropriate choices for f and g) to define a function eval :: Expr -> Int that evaluates an expression to an integer value.

c) Create a function frequencies that, given a string s, creates a list of pairs [(x1,f1) ,....(xk,fk)] such that if the character xi occurs a total number of fi times throughout the list s, then the list of pairs will contain the pair (xi, fi).

As an example of this,

```
frequencies "regninger"
```

should return the list

```
[('r',2),('e',2),('g',2),('n',2),('i',1)]
```

First find out what the type of the function should be.