Mästarprov i Funktionell Programmering
DD1366

Robin Widjeback
robinwid@kth.se

1 Uppgift 1:

1.1 Lösning

```haskell
module Exercise1 where

import Data.Char (toUpper)

-- Function to get the initials of a list of strings
initials :: [String] -> [String]
initials strings = map capitalizeFirstLetter strings

-- Helper function to get the initials of a string
capitalizeFirstLetter :: String -> String
capitalizeFirstLetter str = map toUpperFirst (words
    (replaceHyphens str))

-- Helper function to capitalize the first letter of a
    string and return it
toUpperFirst :: String -> Char
toUpperFirst "" = ' '
toUpperFirst (x:xs) = toUpper x

-- Helper function to replace hyphens with spaces
replaceHyphens :: String -> String
replaceHyphens [] = []
replaceHyphens (x:xs)
  | x == '-' = ' ' : replaceHyphens xs
  | otherwise = x : replaceHyphens xs
```

1.2 Förklaring

initials tar in en lista av strängar och applicerar funktionen capitalizeFirstLetter på varje sträng i listan. Sedan sammanställs resultaten i en lista och returneras.

capitalizeFirstLetter är en hjälpfunktion som tar in en sträng och returnerar en ny sträng. Först används replaceHyphens för att byta ut alla bindestreck mot mellanrum i strängen. Därefter används words för att dela upp strängen i flera strängar, där varje ord separeras av mellanrum. Sedan appliceras toUpperFirst på varje ord för att konvertera den första bokstaven till versal.

toUpperFirst är en hjälpfunktion som tar in en sträng och returnerar den första bokstaven i strängen omgjord till versal. Om strängen är tom returneras ett mellanrumstecken.

replaceHyphens är en hjälpfunktion som tar in en sträng och returnerar en ny sträng där alla bindestreck har blivit utbytta mot mellanrum.

Sammanfattningsvis uppfyller dessa funktioner syftet att extrahera initialerna från en lista med strängar genom att först byta ut bindestreck mot mellanrum, sedan dela upp strängarna i ord, och slutligen konvertera den första bokstaven i varje ord versal och sammanställa dessa versaler i en sträng, vilket görs för varje ord i listan.

2 Uppgift 2:

2.1 Lösning

```haskell
module Exercise2 where

import Data.List (delete)

targetSumPairs :: [Int] -> Int -> ([(Int, Int)], [Int])
targetSumPairs nums target = findPairs nums [] target
    where
        findPairs :: [Int] -> [(Int, Int)] -> Int ->
            ([(Int, Int)], [Int])
        findPairs [] pairs _ = (pairs, [])
        findPairs (x:xs) pairs t =
            let complement = t - x in
                if complement `elem` xs then
                    findPairs (delete complement xs) ((x,
                        complement) : pairs) t
                else
                    let (foundPairs, unmatched) = findPairs
                        xs pairs t in
                        (foundPairs, x : unmatched)
```

2.2 Förklaring

targetSumPairs tar in en lista av Int:s, som vi kallar för nums, och en Int, som vi kallar för target. Funktionen anropar sedan findPairs med nums, en tom lista och target som inputparametrar. Den tomma listan fungerar som vår ackumulator och vi kallar den för pairs.

På rad 11 definierar vi den lokala variabeln complement, som representerar den sökta siffran för att kunna bilda ett par med x. Vi kollar sedan om complementet finns i xs (resten av listan) eller ej. Om det finns, fortsätter vi sökningen med findPairs på xs utan complement och lägger till x och complement i pairs. Target-summan förblir densamma.

Om complementet inte finns i xs, definierar vi den lokala tupeln (foundPairs, unmatched) genom att göra det rekursiva anropet på findPairs med xs, pairs och t som inputvariabler. Vi placerar foundPairs i första delen av tupeln och unmatched tillsammans med x i andra delen av tupeln. Detta betyder att vi fortsätter sökningen med findPairs på xs, och att vi lägger till x i andra listan av tupeln eftersom det inte finns något matchande par för x.

3 Uppgift 3:

3.1 Lösning

```
1 module Exercise3 where
2
3 prefixSums :: [Int] -> [Int]
4 prefixSums list = tail (scanl (+) 0 list)
```

3.2 Förklaring

På rad 4 anropar vi högre ordningens funktion scanl. Scanl tar in en binäroperator, i vårt fall (+), och ett startvärde, i vårt fall 0 och vår inputlista. Scanl summerar det nuvarande värdet med alla tidigare värden i inputlistan och placerar detta nya värde på den nuvarande platsen. Alltså en lista [x0, x1, x2] blir [0, x0, x0+x1, x0+x1+x2]. Med endast denna implementation får vi däremot med vårt startvärde 0, vilket inte är önskvärt, därför tar vi tail av returvärdet från scanl. Tail tar bort det första värdet i en lista. Resultatet blir [x0, x0+x1, x0+x1+x2] vilket var det sökta.

4 Uppgift 4:

4.1 Lösning

```haskell
module Exercise4 where

import Data.Char (toLower)

processStrings :: [String] -> ([String], [String])
processStrings strs =
    let
        isShortOrContainsR :: String -> Bool
        isShortOrContainsR = \str -> length str <= 3 ||
            elem 'r' (map toLower str)
        shortOrContainsR :: [String]
        shortOrContainsR = filter isShortOrContainsR strs
        notShortOrContainsR :: [String]
        notShortOrContainsR = filter (not .
            isShortOrContainsR) strs
    in
        (shortOrContainsR, notShortOrContainsR)
```

4.2 Förklaring

Rad 9 är en predikatfunktion som tar in en sträng och returnerar True om strängens längd är mindre än eller lika med 3 eller om strängen innehåller bokstaven 'r'. Rad 11 lägger alla strängar ur inputlistan som ger True för predikatfunktionen i en lista shortOrContainsR. Rad 13 lägger alla som returnerar False i en annan lista notShortOrContainsR. På rad 15 samanställer vi våra två nya listor i det önskade output-formatet.