Lab 2, Part I.1

Without using recursion or list comprehensions (i.e. using map and foldl), and given the following Prelude functions:

\[
\begin{align*}
\text{const } x \_ &= x \\
\text{flip } f \ x \ y &= f \ y \ x
\end{align*}
\]

code up the following functions:

1. `sumsq` — computes sum of squares of a numeric list
   \[
   \text{sumsq } [1,2,3] = 1 + 4 + 9 = 14
   \]

   \[
   \text{sumsq } = \text{foldl } (+) \ 0 \ . \ \text{map } (\lambda x \rightarrow x \cdot x)
   \]

Lab 2, Part I.2

Without using recursion or list comprehensions (i.e. using map and foldl), and given the following Prelude functions:

\[
\begin{align*}
\text{const } x \_ &= x \\
\text{flip } f \ x \ y &= f \ y \ x
\end{align*}
\]

code up the following functions:

2. `len` — computes length of a list
   \[
   \text{len } [1,2,3] = 3
   \]

   \[
   \text{len } = \text{foldl } (+) \ 0 \ . \ \text{map } (\lambda x \rightarrow 1)
   \]

Lab 2, Part I.3

Without using recursion or list comprehensions (i.e. using map and foldl), and given the following Prelude functions:

\[
\begin{align*}
\text{const } x \_ &= x \\
\text{flip } f \ x \ y &= f \ y \ x
\end{align*}
\]

code up the following functions:

3. `rev` — reverses its list argument
   \[
   \text{rev } [5,6,7,8] = [8,7,6,5]
   \]

   \[
   \text{rev } = \text{foldl } (\text{flip } (:)) \ []
   \]
Lab 2, Part I.4

Without using recursion or list comprehensions (i.e. using map and foldl), and given the following Prelude functions:

\[
\begin{align*}
\text{const } x \_ &= x \\
\text{flip } f \ x \ y &= f \ y \ x \\
\end{align*}
\]

\[
\begin{align*}
f \ . \ g &= x \rightarrow f \ (g \ x) \\
f \$ \ x &= f \ x
\end{align*}
\]

code up the following functions:

4. cat — concatenates two lists together

\[
\text{cat } [1,2,3] \ [4,5,6] = [1,2,3,4,5,6]
\]

\[
\text{cat } l1 \ l2 = \text{foldl} \ (\text{flip} \ (:) ) \ l2 \ (\text{rev} \ l1)
\]

Lab 2, Part II.5

Given a simple binary tree datatype:

\[
\text{data } \text{Tree } a = \text{Nil } | \text{Node } (\text{Tree } a) \ a \ (\text{Tree } a)
\]

Use recursion to:

5. Write a function tmap taking a function \( f \) and tree \( t \) as arguments, that returns the result of applying \( f \) to every value in the tree.

\[
\text{tmap } (+1) \ (\text{Node} \ (\text{Node } \text{Nil} \ 1 \ \text{Nil}) \ 2 \ (\text{Node } \text{Nil} \ 3 \ \text{Nil}))
\]

\[
= (\text{Node} \ (\text{Node } \text{Nil} \ 2 \ \text{Nil}) \ 3 \ (\text{Node } \text{Nil} \ 4 \ \text{Nil}))
\]

\[
\text{tmap } f \ \text{Nil} = \text{Nil}
\]

\[
\text{tmap } f \ (\text{Node} \ \text{left} \ \text{x} \ \text{right})
\]

\[
= \text{Node} \ (\text{tmap } f \ \text{left}) \ (f \ \text{x}) \ (\text{tmap } f \ \text{right})
\]

Lab 2, Part II.6

Given a simple binary tree datatype:

\[
\text{data } \text{Tree } a = \text{Nil } | \text{Node } (\text{Tree } a) \ a \ (\text{Tree } a)
\]

Use recursion to:

6. Write a function tfold taking a function \( f \), value \( v \) and tree \( t \) as arguments, that returns the result of:

- using \( v \) when dealing with an empty tree
- using \( f \) to combine the results of left and right trees with the (middle) node value to get an overall result

\[
\text{myfun } \text{left} \ \text{middle} \ \text{right} = \text{left} + \text{right} + 2 \times \text{middle} 
\]

\[
\text{tfold } \text{myfun} \ 0 \ (\text{Node} \ (\text{Node } \text{Nil} \ 1 \ \text{Nil}) \ 2 \ (\text{Node } \text{Nil} \ 3 \ \text{Nil} ))
\]

\[
= 12
\]

\[
\text{tfold } f \ v \ \text{Nil} = v
\]

\[
\text{tfold } f \ v \ (\text{Node} \ \text{left} \ \text{x} \ \text{right})
\]

\[
= f \ (\text{tfold } f \ v \ \text{left}) \ \text{x} \ (\text{tfold } f \ v \ \text{right})
\]

Lab 3, lab03.hs

\[
\text{-- Lab3 : <Forename> <Surname>}
\]

\[
\text{module Main where}
\]

\[
\text{import Data.Char}
\]

\[
\text{readInteger :: } \text{String} \rightarrow \text{Integer}
\]

\[
\text{readInteger str} = (\text{read} \ str) :: \text{Integer}
\]
Write the following function:

1. calc – reads a file with lines consisting of numbers separated by spaces and outputs a subtotal for each line plus a grand total
   
   calc "file.txt" = "subtotals 23 45 67 - total 135"

You may find these standard functions convenient:

1. lines :: String -> [String]
   breaks a string wherever it encounters a newline character (e.g. \'\n\'), to give a list of lines (with newline removed).
2. unlines :: [String] -> String
   converse of lines, in that it joins up a list of strings with newlines to make one long string
3. words :: String -> [String]
   splits a string at white-space (space, tab, newline)
4. readInteger :: String -> Integer
   converts a string to an integer value, if it can be read as such.

Lab 3, Part I, Solution (a)

```haskell
calc fname
  = do txt <- readFile fname -- read in data from file
       let tottxt = computeTotals txt -- do computation
           putStrLn tottxt -- output result (to stdio)

computeTotals txt
  = reportTotals -- takes list of subtotals and reports
    (map lineSubTotal -- compute subtots for each line
       (lines txt)) -- break txt into lines
```

Lab 3, Part I, Solution (b)

```haskell
lineSubTotal line
  = sum -- sum-up
    (map readInteger -- convert words to integers
       (words line)) -- break line into words

reportTotals stots
  = "subtotals " -- start report with subtotal
    ++ repTot 0 stots -- work through, accumulating total.
  repTot tot []
    = " - total "++show tot -- finish off with total
  repTot tot (stot:stots)
    = show stot ++ " " -- show current subtotal
      ++ repTot (tot+stot) stots -- show rest
```
Write a program `xcl` which implements a simple calculator, via the console

- It maintains a stack of numbers
- If a user enters:
  - a number, it is added to the stack
  - an binary operator, the top two numbers are popped, operated on, and the result is pushed on the stack
  - letter 'c' then the stack is cleared

At each point the stack contents are displayed.

- Hint: think of a recursive function with the stack as an argument.

```haskell
lab3-part2-solution

xcl = xcalc ([]::[Integer]) -- we keep the stack as a parameter

xcalc stk
  = do putStr (show stk ++ " :- ")
  userresp <- getLine
  let stk' = handleCommand userresp stk
      xcalc stk'

handleCommand ('c':_) _ = [] -- clear stack
handleCommand ('+':_) (a:b:rest) = ((a+b):rest)
handleCommand ('-':_) (a:b:rest) = ((a-b):rest)
handleCommand ('*':_) (a:b:rest) = ((a*b):rest)
handleCommand ('/':_) (a:b:rest) = ((a 'div' b):rest)
handleCommand (c:cs) stk
  | isDigit c = (readInteger (c:cs)):stk
  handleCommand _ stk = stk -- otherwise, no change
```