HW 10 **Prof. Caldwell**
Due: 8 October 2013 **COSC 3015**

1 FOldl and Foldr

In class we discussed the *foldl* function which is like *foldr* but which associates to the left.

foldr :: (a → b → b) → b → [a] → b
foldr op id [] = id
foldr op id (x:xs) = x 'op' foldr op id xs

fold :: (b → a → b) → b → [a] → b
foldl op acc [] = acc
foldl op acc (x:xs) = foldl op (acc 'op' x) xs

We have the following example computations.

foldr (+) 0 [1,2,3]
⇝ *1 + (foldr (+) 0 [2,3])*
⇝ *1 + (2 + (foldr (+) 0 [3]))*
⇝ *1 + (2 + (3 + (foldr (+) 0 [])))*
⇝ *1 + (2 + (3 + 0))*
⇝ *1 + (2 + 3)*
⇝ *1 + 5*
⇝ *6*

foldl (+) 0 [1,2,3]
⇝ *foldl (+) (0 + 1) [2,3])*
⇝ *foldl (+) ((0 + 1)+ 2) [3])*
⇝ *foldl (+) (((0 + 1)+ 2) + 3) [])*
⇝ *((0 + 1)+ 2) + 3*
⇝ *(1 + 2) + 3*
⇝ *3 + 3*
⇝ *6*

We discussed that unlike *foldr*, *foldl* is tail-recursive, it does not require the entire list to be processed before it can start accumulating the result. Tail recursive functions are more efficient because they do not use up stack space storing the partial results waiting to be evaluated (as can be seen in the example with *foldr* above.

To really guarantee efficiency, use *fold'* which forces evaluation of the accumulated value before futher unfoldting the recursive call.

foldl' (+) 0 [1,2,3]
⇝ *foldl' (+) (0 + 1) [2,3])*
⇝ *foldl' (+) (1) [2,3])*
⇝ *foldl' (+) (1 + 2) [3])*
⇝ *foldl' (+) (3) [3])*
⇝ *foldl' (+) (3 + 3) [])*

$\rightsquigarrow$ *foldl' (+) (6) [])*
$\rightsquigarrow$ *6*

A recursive function is tail recursive if the final result of the recursive call is the final result of the function itself. If the result of the recursive call must be further processed (say, by adding 1 to it, or consing another element onto the beginning of it), it is not tail recursive. This definition may be a little opaque, but the main thing to notice is that the topmost function in the recursive call in the definition of *foldl* is to the function *foldl* itself. A nice discussion an more fomal defintion can be found at `http:///www.haskell.org/haskellwiki/Tail_recursion`.

Most recursive functions on lists can be made tail recursive by using an accumulator.

For example, the following natural definition of the *len* function is not tail recursive.

```
len :: [a] → Int
len [] = 0
len (x:xs) = 1 + len xs
```

We can make a tail recursive version (which has a slightly different type) by adding an accumulator as follows:

```
len_acc :: Int → [a] → Int
len_acc k [] = k
len_acc k (x:xs) = len_acc (k + 1) xs
```

Then *len* can be defined in terms of *len_acc*:

$$len = len\_acc\ 0$$

We can define a tail recursive version of *len* hiding the *len_acc* version using a where clause as follows:

```
len :: [a] → Int
len = len_acc 0
   where len_acc k [] = k
            len_acc k (x:xs)  = len_acc (k + 1) xs
```

We can use a where clause to hide the accumulator by making it local to the definition.

```
len' :: [a] → Int
len' xs = len_acc xs 0
   where len_acc [] acc = acc
            len_acc (x:xs) acc = len_acc xs (1 + acc)
```

Problem 1.1. Write a tail recursive version of the *filter* function using an accumulator. The non-tail recursive verions is as follows:

filter :: (a → Bool) → [a] → [a]
filter p [] = []
filter p (x:xs) = if (p x) then x : filter p xs else filter p xs

Problem 1.2. Write a tail recursive version of the *rev* function using an accumulator. The non-tail recursive verions is as follows:

rev :: [a] → [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]

Folds are rich - they can be used to implement other list functions. For example, we can implement reverse using `foldr` and `foldl` as follows.

revr = foldr (\y ys → ys ++ [y]) []
revl = foldl (flip (:)) []

Problem 1.3. Use *foldl* (or *foldr* to implement *map* and *filter*.