Lists.hs

module Lists where

-- we're redefining many functions in the standard Prelude, so we
-- suppress importing them here
import Prelude hiding ( length, sum, product, reverse, minimum )

-- Compute the length of a list
length :: [a] -> Integer
length [] = 0
length (_:xs) = 1 + length xs

-- Compute the sum of a list of Integers
sum :: [Integer] -> Integer
sum [] = 0
sum (x:xs) = x + sum xs

-- Compute the product of a list of Integers
product :: [Integer] -> Integer
product [] = 1
product (x:xs) = x * product xs

-- Increment all Integers in a list by a value
incBy :: [Integer] -> Integer -> [Integer]
incBy [] _ = []
incBy (x:xs) n = x + n : incBy xs n

-- Update the nth Integer in a list with a new value, if the list is
-- long enough. (Does nothing otherwise.)
updateByIndex :: Integer -> Integer -> [Integer] -> [Integer]
updateByIndex _ _ [] = []
updateByIndex 0 y (x:xs) = y : xs
updateByIndex n y (x:xs) = x : updateByIndex (n-1) y xs

-- Append two lists
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys

-- Add an element to the end of a list.
snoc :: [a] -> a -> [a]
snoc [] y = [y]
snoc (x:xs) y = x : snoc xs y

-- Reverse the elements in a list
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = snoc (reverse xs) x

-- This runs in O(n^2) time where n is the length of the list; why?

-- Reverse the elements in a list in linear time
fastReverse :: [a] -> [a]
    -- This uses "where" to define a local variable
fastReverse xs = go [] xs
    where
        go acc [] = acc
        go acc (x:xs) = go (x:acc) xs

-- Compute both the quotient and modulus of two numbers
divMod :: Integer -> Integer -> (Integer, Integer)
divMod n d = (n `div` d, n `mod` d)

-- Split a list into evens and odds
evensOdds :: [Integer] -> ([Integer], [Integer])
evensOdds [] = ([], [])
evensOdds (x:xs) = (x:evens, odds)
  | even x    = (x:evens, odds)
  | otherwise = (evens, x:odds)
  where  -- a "where" scopes over multiple guarded equations
         (evens, odds) = evensOdds xs

divide one number by another, returning Nothing if the divisor is 0
safeDiv :: Integer -> Integer -> Maybe Integer
  -- "Maybe" offers the possibility of failure, called Nothing
  -- success is called Just
  safeDiv _ 0 = Nothing
  safeDiv n d = Just (n \div d)

-- Retrieve the first element of a list, if one exists
safeHead :: [a] -> Maybe a
  safeHead [] = Nothing
  safeHead (x:_)? = Just x

-- Retrieve the tail of a list, if it exists
safeTail :: [a] -> Maybe [a]
  safeTail [] = Nothing
  safeTail (_:xs) = Just xs

-- Compute the minimum of a list, if it exists
minimum :: Ord a => [a] -> Maybe a
  -- The "Ord a =>" constraint says that the type a is ordered.
  -- See what happens if you leave it out!
  minimum [] = Nothing
  minimum (x:xs) = Just (go x xs)
  where
    go x [] = x
    go x (y:ys)
      | x < y = go x ys
      | otherwise = go y ys

-- Tests whether or not the argument equals itself.
isReflexive :: Eq a => a -> Bool
  isReflexive x = x == x