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    -- n
               -> Integer    -- new value
               -> [Integer]  -- original list
               -> [Integer]
updateByIndex _ _ []     = []      -- we've run out of elements
updateByIndex 0 y (_:xs) = y : xs  -- ignore old value, use y instead
updateByIndex n y (x:xs) = x : updateByIndex (n-1) y xs  -- recur

-- 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]
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 [] = ([], [])
Lists.hs

73:  evensOdds (x:xs)
74:    | even x = (x:evens, odds)
75:    | otherwise = (evens, x:odds)
76: where -- a "where" scopes over multiple guarded equations
77:      (evens, odds) = evensOdds xs
78:    
79: -- Divide one number by another, returning Nothing if the divisor is 0
80:  safeDiv :: Integer -> Integer -> Maybe Integer
81:    -- "Maybe" offers the possibility of failure, called Nothing
82:    -- success is called Just
83:  safeDiv _ 0 = Nothing
84:  safeDiv n d = Just (n `div` d)
85:    
86: -- Retrieve the first element of a list, if one exists
87:  safeHead :: [a] -> Maybe a
88:  safeHead [] = Nothing
89:  safeHead (x:_)= Just x
90:    
91: -- Retrieve the tail of a list, if it exists
92:  safeTail :: [a] -> Maybe [a]
93:  safeTail [] = Nothing
94:  safeTail (_:xs) = Just xs
95:    
96: -- Compute the minimum of a list, if it exists
97:  minimum :: Ord a => [a] -> Maybe a
98:    -- The "Ord a =>" constraint says that the type a is ordered.
99:    -- See what happens if you leave it out!
100: minimum [] = Nothing
101: minimum (x:xs) = Just (go x xs)
102:   where
103:      go x [] = x
104:      go x (y:ys)
105:        | x < y = go x ys
106:        | otherwise = go y ys
107:    
108: -- Tests whether or not the argument equals itself.
109: isReflexive :: Eq a => a -> Bool
110: isReflexive x = x == x