Currying

a) fun filter_by_example f x = 
    List.filter (fn x' => f x = f x')

b) fun same_size_as xs = filter_by_example List.length xs

c) fun count_o s = 
   List.length (List.filter (fn x => x = "o") (String.explode s))

   fun count_o s = 
       List.foldl (fn (x, acc) => if x = "o" then acc + 1 else acc) 
       0 (String.explode s)

d) val silly_application = filter_by_example count_o "dogsarecool"

e) fun contains x = 
    List.foldl (fn (x', acc) => acc orelse x' = x) 
    false

f) fun filter_unique f prev xs = 
   case xs of 
        [] => [] |
        x'::xs' =>
            let val result = f x'
in

if contains result prev
then filter_unique f prev xs'
else x' :: filter_unique f (result :: prev) xs'
end

g) fun unique_sums xs = filter_unique List.length [] xs

h) fun all_that_contain x = (List.filter (contains x))

i) val even_only =
   List.map (List.filter (fn x => x mod 2 = 0))

j) fun even_only_not_empty xs =
   List.filter (not o List.null) (even_only xs)

Modules
a)

signature RATIONAL =
sig
  type rational
  exception BadFrac
  
  val make_frac : int * int -> rational
  val toString : rational -> string
  val add : rational * rational -> rational
  val Whole : int -> rational

end

structure Rational => RATIONAL =
struct
  type rational = int * int
  exception BadFrac

  fun toString (x,y) = (* to_string body *)
  fun Whole i = (i,1)
  fun make_frac (x,y) = (* make_frac body *)
  fun add ((a,b),(c,d)) = (* add body *)

end