exception ListLengthMismatch

(*
'a list * 'b list * 'c list -> ('a * 'b * 'c) list
('a * 'b * 'c) list -> 'a list * 'b list * 'c list
*)

(* don't do this *)
fun old_zip3 (l1,l2,l3) =
  (* 'a list * 'b list * 'c list -> ('a * 'b * 'c) list ('a * 'b * 'c) list -> 'a list * 'b list * 'c list*)(* don't do this *)
  fun multsign (x1, x2) =
    if null l1 
    ∧ 
    null l2 
    ∧ 
    null l3 
    then 
    [] 
    else 
    if null l1 
    ∨ 
    null l2 
    null l3 
    then 
    P 
    else 
    if x=0 
    then 
    Z 
    else 
    if x>0 
    then 
    P 
    else 
    N 
    case (sign x1,sign x2) 
    fun shallow_zip3 (l1,l2,l3) =
      (* don't do this *)
      fun sign x =
        if x=0 
        then 
        Z 
        else 
        if x>0 
        then 
        P 
        else 
        N 
        case
      fun len xs =
        (* simpler use of wildcard pattern when you do not need the data *)
        case
      fun zip3 list_triple =
        (* do this *)
        fun unzip3 lst =
          (* and the inverse *)
          fun nondecreasing xs =
            (* another elegant use of "deep" patterns *)
          fun cumulative_sum xs =
            (* or remember this from homework 1 without pattern-matching? *)
          fun fact1 n =
            (* nested pattern-matching often convenient even without recursion; also the wildcard pattern is good style 

Winter 2018  CSE 341: Programming Languages  1/2
fun fact2 n = 
  let fun aux(n, acc) = if n=0 then acc else aux(n-1, acc*n) 
  in aux(n, 1) 
  end

(* more examples of making functions tail-recursive *)

fun sum1 xs = 
  case xs of 
    [] ⇒ 0 
  | i::xs' ⇒ i + sum1 xs'

fun sum2 xs = 
  let fun f (xs, acc) = 
    case xs of 
      [] ⇒ acc 
    | i::xs' ⇒ f(xs', i+acc) 
  in f(xs, 0) 
  end

fun rev1 xs = 
  case xs of 
    [] ⇒ [] 
  | x::xs' ⇒ (rev1 xs') @ [x]

fun rev2 xs = 
  let fun aux(xs, acc) = 
    case xs of 
      [] ⇒ acc 
    | x::xs' ⇒ aux(xs', x::acc) 
  in aux(xs, []) 
  end