exception ListLengthMismatch

(* don’t do this *)

fun old_zip3 (11,12,13) =
  if null 11 ∧ null 12 ∧ null 13
  then []
  else if null 11 v null 12 v null 13
  then raise ListLengthMismatch
  else (hd 11, hd 12, hd 13) :: old_zip3(tl 11, tl 12, tl 13)

(* don’t do this *)

fun shallow_zip3 (11,12,13) =
  case l1 of
    [] ⇒ []
  | [] ⇒ raise ListLengthMismatch
  | _ ⇒ raise ListLengthMismatch
  | hd1::tl1 ⇒ (hd1,hd2,hd3)::shallow_zip3(tl1,tl2,tl3))

(* do this *)

fun zip3 list_triple =
  case list_triple of
    ([],[][]) ⇒ Int.max(x,maxlist(xs',ex))
  | _ ⇒ raise ex
  | x::[] ⇒ x
  | x::xs′ ⇒ Int.max(x,maxlist(xs′,ex))
  | (a,b,c) :: tl ⇒ let
      val l1 = unzip3 tl
    in
      (a,b,c)::zip3(tl1,tl2,tl3)
    end

(* and the inverse *)

fun unzip3 lst =
  case lst of
    [] ⇒ []
  | (a,b,c)::tl ⇒ let
      val v = maxlist ([a,b,c],MyUndesirableCondition)
    in
      (a,b,c)::unzip3 tl
    end

(* another elegant use of "deep" patterns *)

fun nondecreasing xs =
  case xs of
    [] ⇒ true
  | x::[] ⇒ true
  | (head::neck::rest) ⇒ (head ≤ neck ∧ nondecreasing (neck::rest))

(* or remember this from homework 1 without pattern-matching? *)

fun cumulative_sum xs =
  case xs of
    [] ⇒ xs
  | x::[] ⇒ x
  | (head::neck::rest) ⇒ head :: cumulative_sum ((head+neck) :: rest)

(* nested pattern-matching often convenient even without recursion; also the wildcard pattern is good style match on a pair and one or more parts of it quite useful on homework 2 *)

datatype sgn = P | N | Z

fun multisign (x1,x2) =
  let fun sign x = if x=0 then P else if x>0 then P else N
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