fun foldl (f: α×β→β) (acc: β) (l: α list): β =
  case l of
    [] ⇒ acc
  | x::xs ⇒ foldl f (f(x,acc)) xs

fun foldr (f: α×β→β) (acc: β) (l: α list): β =
  case l of
    [] ⇒ acc
  | x::xs ⇒ f(x, (foldr f acc xs))

val numbers = [1, 1, 2, 3, 4, 5, 6, 7, 9, 13, 11, 81]
val is_valid (x, y) = (x + y = 17)
val all_pairs = List.map (λ x ⇒ List.map (λ y ⇒ (x, y)) numbers) numbers
val valid_pairs = List.map (List.filter is_valid) all_pairs
val non_empty = List.filter (λ x ⇒ ¬(null x)) valid_pairs
val flattened = List.foldl (λ (x, acc) ⇒ (x @ acc)) [] non_empty

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

fun is_alternating_v1 (1::tail) = zero_v1 tail
| is_alternating_v1 (0::tail) = one_v1 zero_v1 tail
| is_alternating_v1 [] = true
| is_alternating_v1 _ = false

fun is_alt xs = one_v1 zero_v1 xs
| zero_v1 _ = false

fun one [] = true
| one (1::tail) = zero tail
| one _ = false

and zero [] = true
| zero (0::tail) = one tail
| zero _ = false

fun is_alternating (1::tail) = zero tail
| is_alternating (0::tail) = one tail
| is_alternating [] = true
| is_alternating _ = false

fun is_alt' xs = one xs ∨ zero xs

(* There are a lot of ways to flatten a list of lists into one list. *)

(* How would we flatten the list with flat_map? *)

val flatten_with_flat_map = flat_map (λ x ⇒ x)

(* Here we can use foldr *)

fun is_alternating xs = one_v1 zero_v1 xs
| zero_v1 _ = false

| one_v1 _ = false
| one_v1 [] = true

and zero_v1 _ = false

fun is_alternating (1::tail) = zero_v1 tail
| is_alternating (0::tail) = one_v1 zero_v1 tail
| is_alternating [] = true
| is_alternating _ = false

fun is_alt_prime xs = one_v1 zero_v1 xs

(* We don’t need this syntax though *)

(* How might we do this without it? *)

(* Notice that every call is a tail call *)

(* turns out we can do this for any finite state machine *)