Section Agenda

- Homework 3 due Monday... any questions?
- Midterm in class May 3rd (a week from Friday).
- Mutual recursion
- Modules
- Higher-order functions and Currying practice
Want to write a function that takes a list and returns a bool which is true iff the list has alternating 0s and 1s.

```
val is_alternating = fn : int list -> bool

• is_alternating [0,1,0] = true
• is_alternating [1,0,1] = true
• is_alternating [1,1,0] = false
```
A (first) solution sketch

One idea:

- `val zero = fn : int list -> bool`
- `val one = fn : int list -> bool`

Start in either one function or the other, return `true` iff the list begins with a zero or one, and then recur on the other.
A problem

fun zero [] = true
  |
  | zero 0::xs' = one xs'
  |
  | _ = false

fun one [] = true
  |
  | one 1::xs' = zero xs'
  |
  | _ = false
Mutual recursion

fun zero [] = true
  
  | zero 0::xs' = one xs'
  
  | _ = false

and one [] = true
  
  | one 1::xs' = zero xs'
  
  | _ = false
A solution

fun zero [] = true
  | zero 0::xs' = one xs'
  | _ = false

and one [] = true
  | one 1::xs' = zero xs'
  | _ = false

fun is_alternating [] = true
  | is_alternating 0::xs' = one xs'
  | is_alternating 1::xs' = zero xs'
  | _ = false
An (alternative) solution

fun zero [] = true
    | zero 0::xs' = one xs'
    | _ = false
and one [] = true
    | one 1::xs' = zero xs'
    | _ = false

fun is_alternating xs = case xs of
    [] => true
    | 0::xs' => one xs'
    | 1::xs' => zero xs'
    | _ => false
Modules

- Good for organization and managing namespaces
  - Can organize bindings into separate modules so that everything is not at the top level

- Good for maintaining invariants
  - Maintain invariants within a module by hiding implementation details from a client
Modules - Examples of Invariants

- **Ordering of operations**
  - e.g. restrict to insert, then query

- **Data kept in good state**
  - e.g. keep fractions simplified (RATIONAL example from lecture!)

- **Policies followed**
  - e.g. don’t allow shipping request without purchase order
In lecture we saw this example of a module:

```ml
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
val doubler : int -> int
end

structure MyMathLib :> MATHLIB =
struct
fun fact x = ...
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
```
In lecture we saw this example of a module:

What happens if we remove this line from the signature?

```plaintext
signature MATHLIB =
  sig
  val fact : int -> int
  val half_pi : real
  val doubler : int -> int
  end

structure MyMathLib :> MATHLIB = struct
  fun fact x = ...
  val half_pi = Math.pi / 2.0
  fun doubler x = x * 2
  end
```
Practice with modules..!
Write a function that takes an int list and produces an (int * int) list which contains all pairs of elements in the original list.

```
val all_pairs = fn : int list -> (int * int) list
```
Higher-order practice #2

Now let's say we want only pairs which are either (even, odd) or (odd, even) (but not (even, even), etc.).

```plaintext
val all_even_pairs xs = fn : int list -> (int * int) list
```