Nice write-up.

```
// ----------------------------------------------------------------
//        Homework 1
//        Adam Levy, Alejandro Palacios, & Alicia Rodriguez
//        September 19, 2016
// ----------------------------------------------------------------


// 1. A fraction like 2/3 can be represented in F# as a pair of type int * int.
// Define infix operators .+ and .* to do addition and multiplication of
// fractions:

(*
   > (1,2) .+ (1,3);;
   val it : int * int = (5, 6)
   > (1,2) .+ (2,3) .* (3,7);;
   val it : int * int = (11, 14)
*)

// Note that the F# syntax for defining such an infix operator looks like this:

(*
   let (.+) (a,b) (c,d) = ...
*)

// Also note that .+ and .* get the same precedences as + and *, respectively,
// which is why the second example above gives the result it does.
// Finally, note that your functions should always return fractions in lowest
// terms.
// To implement this, you will need an auxiliary function to calculate
// the gcd (greatest common divisor) of the numerator and the denominator;
// this can be done very efficiently using Euclid's algorithm, which can be
// implemented in F# as follows:

let rec gcd = function
| (a,0) -> a
| (a,b) -> gcd (b, a % b);;

// Solution:

(*
  Adding Fractions Function
  The function multiplies the numerators by the opposite fraction's
  denominators. This is done because the denominators will also be multiplied
```

by each other so they can be the same. Once multiplied, the numerators are added together in the first tuple and divided by the gcd. The denominators are multiplied in the second tuple since the fraction needs to have a common denominator. For the gcd, the calculations are repeated so that the gcd can be found for the two fractions. Once the gcd is found, the numerator and denominator are divided by the gcd so that the fraction can be returned in lowest terms. *)

It would be better not to compute a*d+b*c twice.

```
let (.+) (a,b) (c,d) =
  let gcdResult = gcd(a * d + c * b , b * d)
  (a * d + c * b / gcdResult, b * d / gcdResult);;
```

```
(*
  Multiplying Fractions Function
  The numerators are multiplied together and divided by the gcd. The same
  process is done in the second tuple for the denominator. To find the gcd,
  the calculations are repeated for the gcd function. The numerator and
  denominator are each multiplied by the gcd to return the function in lowest
  terms. *)

let (.*) (a,b) (c,d) =
  let gcdResult = gcd (a * c, b * d)
  (a * c / gcdResult, b * d / gcdResult);;
```

Note that you don't need to terminate your definitions with ;;

```
// 2. Write an F# function revlists xs that takes a list of lists xs and
// reverses all the sub-lists:

(*
    > revlists [[0;1;1];[3;2];[];[5]];;
    val it : int list list = [[1; 1; 0]; [2; 3]; []; [5]]
*)

// Hint: This takes just one line of code, using List.map and List.rev.

// Solution:

(* List.map is called on the List.rev function, which takes a list xs on a
   parameter. List.rev is mapped to every list element of the list list. *)

let revlists xs = List.map List.rev xs;;
```

// 3. Write an F# function interleave(xs,ys) that interleaves two lists:

```
(*
    > interleave ([1;2;3],[4;5;6]);;
    val it : int list = [1; 4; 2; 5; 3; 6]
*)
```

// Assume that the two lists have the same length.

// Solution:

```
(*
  Assuming the lists are of the same length, the base case returns an empty list
  if the first list is empty. The recursive case uses the cons operation on ys
  with the next iteration of the function. This is then consed with xs. It is
  important to note that not all cases are covered. However, since we can assume
  both lists are the same length they don't need to be. To avoid the warning you
  can replace the base case with the following base cases:
  |(x::xs, []) -> [x]
  |([], y::ys) -> [y]
  |([], []) -> []
*)
let rec interleave = function
  | ([],[]) -> []
  | (x::xs, y::ys) -> x::y::interleave(xs,ys);;
```

(Here an exception would be preferable.)

```
// 4. Write an F# function cut xs that cuts a list into two equal parts:

(*
   > cut [1;2;3;4;5;6];;
   val it : int list * int list = ([1; 2; 3], [4; 5; 6])
*)

// Assume that the list has even length.
// To implement cut, first define an auxiliary function gencut(n, xs) that
// cuts xs into two pieces, where n gives the size of the first piece:

(*
   > gencut(2, [1;3;4;2;7;0;9]);;
   val it : int list * int list = ([1; 3], [4; 2; 7; 0; 9])
*)

// Paradoxically, although gencut is more general than cut, it is easier to
// write! (This is an example of Polya's Inventor's Paradox: "The more
// ambitious plan may have more chances of success.")

// Another Hint: To write gencut efficiently, it is quite convenient to use
// F#'s local let expression (as in the cos_squared example in the Notes).

// Solution:

(*
  Gencut Auxillary Function with pattern matching syntax
  The base case will return a tuple with an empty list and the list you wished
  to cut. The recursive case defined an inner tuple that will apply the gencut
  function to the tail of the list and decrement the index by 1. The head of the
  list is consed with the first list i (the one that will return all of the
  elements up to the index), and the remainder of the list is returned as j.

  Note: We do not need to worry about the missing cases because we assume that
  we will not need to cut the list at all if it is empty.
*)
let rec gencut = function
  | (0, xs) -> ([], xs)
  | (n, x::xs) -> let (i, j) = gencut(n-1, xs)
                  (x::i, j);;

(*
  Cut Function
```

```
  Gets the length of the half of the list and passes it into gencut to return
  the list split into two halves.
*)
let cut xs =
  let half = List.length xs / 2
  gencut(half, xs);;
```

```
// 5. Write an F# function shuffle xs that takes an even-length list, cuts it
// into two equal-sized pieces,
// and then interleaves the pieces:

(*
   > shuffle [1;2;3;4;5;6;7;8];;
   val it : int list = [1; 5; 2; 6; 3; 7; 4; 8]
*)

// (On a deck of cards, this is called a perfect out-shuffle.)

// Solution:

let shuffle xs = interleave(cut xs);;
```

```
// 6. Write an F# function countshuffles n that counts how many calls to shuffle
// on a deck of n distinct "cards" it takes to put the deck back into its
// original order:

(*
   > countshuffles 4;;
   val it : int = 2
*)

// (To see that this result is correct, note that shuffle [1;2;3;4] = [1;3;2;4],
// and shuffle [1;3;2;4] = [1;2;3;4].)
// What is countshuffles 52?

// Hint: Define an auxiliary function countaux(deck, target) that takes two
// lists and returns the number of shuffles it takes to make deck equal to
// target.

// Solution:

(*
  Countaux Auxillary Function
  Receives a deck and returns how many shuffles that deck will take for it to
  match the target deck.
*)
let rec countaux (deck, target) =
  if deck = target then 0 else 1 + countaux(shuffle deck, target);;

(*
  Countshuffles Function
  Received a number and will get a list from [1..n]. It will then see how many
  shuffles it takes to get back to the original list.

  Ex: [1;2;3;4] will be shuffled into [1;3;2;4] and then will be shuffled again
  to return [1;2;3;4], the original list. This took 2 shuffles in total. For a
  deck of 52 cards, it will take 8 shuffles to return it back to [1..52].
*)
let countshuffles n =
  let original = [1..n]
  1 + countaux(shuffle original, original);;
```