This lecture includes some larger examples in F# and introduces higher-functions.
Examples: append

- Now that we have lists, we’ll see some examples of typical functional programming.
- Along the way we’ll see some other features of F#.
- First example - concatenating two lists into one:

```fsharp
let rec append xs ys =
    match xs with
    | [] -> ys
    | x::xxs -> x :: append xxs ys
```

- Here we have a function that takes two arguments.
- This is similar to taking a pair, but is really a kind of higher-order function (later).
- This function takes the first argument apart via pattern matching, recursively appends the tail, then adds the head.
- Appending is such a common operation it is in the F# core library as the infix function @. E.g.

```
[1,2,3] @ [4,5]
val it : int list = [1,2,3,4,5]
```
Examples: qsort

- The following function sorts a list using a recursive quicksort.

```ml
let rec qsort = function
    | [] -> []
    | x1::xs -> qsort [for x in xs do if x<=x1 then yield x]
         @ [x1]
         @ qsort [for x in xs do if x>x1 then yield x]
```

- This is efficient algorithm – on average $O(n \log n)$.
- We use `@` to concatenate three lists (on separate lines).
  - The result of sorting all $x$ less than or equal to the first one.
  - Then, just the first element.
  - Then the result of sorting the remaining elements.
- List comprehensions allow us to extract sub-lists easily.
Examples: reverse

- The following function reverses a list.

```ocaml
let reverse xs =
  let rec revAppend = function
    | (acc, []) -> acc
    | (acc, x :: xxs) -> revAppend (x :: acc, xxs)
  revAppend ([], xs)
```

- The local function `revAppend` takes two arguments as a pair.
  - It reverses the 2nd argument, placing the result in front of `acc`, the 1st arg.
  - Reversing is done by adding the incrementally adding the head to `acc`.
  - This is a common pattern called an *accumulator argument*.

- This function does not use extra memory for longer lists even though it calls itself once for each element in the list.
  - This is because the result of the recursive call is returned directly.
  - So, the current variables can be discarded before the recursive call.
  - Then, the recursive call directly returns a result to the original caller.
  - The compiled code will actually just be a simple loop.
  - This is called *tail recursion*, and is a very common technique for efficient code.

- The reverse function is in the core libraries as `List.rev`
An inefficient version of reverse

- For comparison, here is a simpler implementation of reverse that avoids using an accumulator, but is slow: $O(n^2)$

  ```haskell
  let revSlow = function
  | [] -> []
  | (x :: xs) -> (reverse xs) @ [x]
  ```

- It is slow because @ (append) rebuilds its first argument, which is on average half the size of the original list.
  - This is done once for each element, hence the $O(n^2)$.
  - `revSlow [1,2,..,n-1,n]` does ...(([] @ [n]) @ [n-1]) @ ... @ [1]

- In general, you should be careful when using @.
  - Avoid applying @ many times with large first arguments.
  - An accumulator builds up the result at function calls, rather than at returns.
  - `reverse [1,2,..,n-1,n]` does [n] :: ... ([2] :: ([1] :: []))
F# core libraries

- F# comes with its own libraries, and can also access .NET libraries.
- You can access the documentation for the libraries via the start menu (in the F# CTP) – see List, Core, Math, ...
- Integration with .NET is tight – e.g., lists implement the .NET interface for Enumerable collections.
- We’ll see various functions from the libraries as we go.
- When we come to concurrency, we’ll increasingly make use of .NET libraries.
- The F# libraries are specifically designed for functional programming.
  - They avoid modifying data structures, and instead produce new ones that use sharing.
Higher-order functions

- Higher-order functions are functions that take functions as arguments or return functions.
  - This includes functions within data structures.

- We have actually already seen some – our definition of append is actually short hand for:

```ml
let rec append = fun xs ->
    (fun ys ->
     match xs with
     | [] -> ys
     | x::xs -> x :: append xs ys
    )
```

- Here `fun` constructs a function
  - `(function` does the same but with many patterns.)

- `append` is actually a function that returns a function!
Example: map

- The following is an example of a function that takes a function as an argument.

  ```
  let rec map f = function
  | [] -> []
  | x::xs -> (f x) :: (map f xs)
  ```

- [Note that the parens are not needed, and generally we’ll leave them out in future.]

- This function applies another function to every element in a list.

- E.g., we can do:

  ```
  > map reverse [[1,2,3], [4,5,6]];;
  val it : int list list = [[3, 2, 1], [6, 5, 4]]
  ```